HZ BOOKS
华章科技

Linux/Unix
技术丛书

Linux内核探秘

深入解析文件系统和设备驱动的架构与设计

高剑林 著

Explore Linux Kernel
In-depth Analysis of Architecture and Design about File System and Device Driver

- 腾讯资深Linux内核专家10余年工作经验结晶，业界多位专家联袂推荐，Linux内核工程师和驱动开发工程师的必读之作
- 从工业需求角度另辟蹊径，注重效率和实用性，将Linux内核分为基础部分和应用部分以及内核架构和内核实现两个维度，对Linux内核的文件系统、设备驱动的架构设计与实现原理进行了深入分析



机械工业出版社
China Machine Press

剑林是一位资深的Linux内核专家，对Linux内核的机制有很深入的了解。本书是他10余年工程经验的结晶，以一个技术开发人员的视角，由浅入深，诠释了内核的文件系统和设备驱动的内部机制。不管是对内核感兴趣的初学者，还是内核研发人员，都可以从本书中获益匪浅！

——王营　百度高级研发工程师

本书由国内著名互联网企业的一线架构师撰写。作者在工作中能够快速地制定出系统方案，与作者对操作系统内核的深入理解密不可分。本书基础层和应用层的划分颇具新意，能很好地帮助读者理清Linux内核的脉络，希望有更多人学习和研究系统内核，提升国内企业的核心能力。

——谢宝友　中兴通讯操作系统平台部技术总工程师

作为长期从事内核和分布式文件系统开发的技术人员，我很高兴看到我的朋友高剑林出版这本书。从本书的结构和内容来看，这是一本授之以渔的书。虽然Linux内核代码庞杂，但这本书却能为读者迅速理出一条清晰的主线，让读者感受到Linux内核的魅力并继续深入进去。非常值得一读，推荐！

——许家强　IBM高级工程师

Linux的内核也是Android的内核，理解Linux内核是Android系统级研发人员的必备技能之一。相对于传统的Linux经典专著，剑林这本书更多地从实践和应用的视角分析了内核的架构和设计，既可以为读者展示内核的清晰脉络，又不至于让读者迷失在Linux的汪洋大海中。

——杨云君　SONY资深Android系统专家/ 畅销书《Android的设计与实现：卷I》作者

和剑林是多年同事，他一直专注于内核子系统各个层次的研究，现在转而从事分布式文件系统的研究。本书读起来比其他内核的书籍更加亲切，充分体现了剑林深厚的工程开发能力，绝对是操作系统入门必不可少的首选书。书中内容通俗易懂，入门门槛低。

——孙子荀　腾讯高级工程师



上架指导：计算机/程序设计/Linu

ISBN 978-7-111-44585-2



定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991　88361066
购书热线：(010) 68326294　88379649　68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

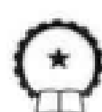
Linux内核探秘

深入解析文件系统和设备驱动的架构与设计

Explore Linux Kernel

In-depth Analysis of Architecture and Design about File System and Device Driver

高剑林 著


机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Linux 内核探秘：深入解析文件系统和设备驱动的架构与设计 / 高剑林著 .—北京：机械工业出版社，2013.12
（Linux/UNIX 技术丛书）

ISBN 978-7-111-44585-2

Ⅰ. L… Ⅱ. 高… Ⅲ. Linux 操作系统 Ⅳ. TP316.89

中国版本图书馆 CIP 数据核字（2013）第 256030 号

本书从工业需求角度出发，注重效率和实用性，是帮助内核研发及调试、驱动开发等领域工程师正确认识并高效利用 Linux 内核的难得佳作！作者是腾讯公司资深的 Linux 内核专家和存储系统专家，在该领域工作和研究的 10 余年间，面试了数百位 Linux 内核工程师，深知学习 Linux 内核过程中经常遇到的困惑，以及在工作中容易犯的错误。基于这些原因作者撰写了本书。本书出发点和写作方式可谓独辟蹊径，将 Linux 内核分为两个维度，一是基础部分和应用部分，二是内核架构和内核实现，将两个维有机统一，深入分析了 Linux 内核的文件系统、设备驱动的架构设计与实现原理。

全书在逻辑上分为三部分：第一部分（第 1 ~ 2 章）首先将内核层划分为基础层和应用层，讲解了基础层包含的服务和数据结构，以及应用层包含的各种功能，然后对文件系统的架构进行了提纲挈领的介绍，为读者学习后面的知识打下基础；第二部分（第 3 ~ 9 章）从设备到总线到驱动，逐步深入，剖析了设备的总体架构、为设备服务的特殊文件系统 sysfs、字符设备和 input 设备、platform 总线、serio 总线、PCI 总线、块设备的实现原理和工作机制；第三部分（第 10 ~ 13 章）对文件系统的读写机制进行了深入分析，最后通过一个真实文件系统 ext2，复习本书所有知识点。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：白　宇
北京市荣盛彩色印刷有限公司印刷
2014 年 1 月第 1 版第 1 次印刷
186mm × 240 mm • 14.5 印张
标准书号：ISBN 978-7-111-44585-2
定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

前　言

为什么要写这本书

目前市面上关于 Linux 内核方面的书籍可以分为两类，一类是学院派书籍，其中比较有名的包括《深入理解 Linux 内核》(ULK）等；一类是国内特有的培训教材。大体而言，学院派的书籍体系一般很完整，在广度和深度上都有完善的阐述。可惜也正因为它的完整、复杂和庞大，使得阅读学院派的书籍往往是个艰巨的任务。ULK 这本书已经有八百页的体量，还有很多细节知识没有讲述，而早期的《内核情景分析》一书更是达到上千页的体量。以至于业界公认内核的学习曲线最陡峭，学习难度最大。

而本书是从工业界角度出发，为工业界使用而写。比较关注计算机科学方面进展的工程师，应可以意识到计算机科学和计算机工业是两个不同的领域。前者注重创新和理论完备，后者注重效率和实用。从效率和实用的角度，需要在降低学习难度的基础上提供相对体系化的结构，这就必须对庞大复杂的内核进行分解和抽取，这也正是本书试图将内核分解为基础层和应用层的原因。

这些年来，笔者先后面试过上百位内核工程师，组织过多次讲座或者交流会议，和国内多家公司的一流工程师有过深入交流。总体而言，国内内核应用和开发的水平处于非常低的水平，这一方面表现在理解内核的技术人员在国内总体上不多，即使是专业的内核的工程师，对内核的一些基本问题理解不清甚至理解错误的也不在少数；另一方面是大多数人认为内核在工作中用处不大，很难发挥价值。

针对第一个问题，笔者做过调查问卷，通过调查发现，公认学习内核最大的问题就是内核代码的难懂和跳跃。从一个函数跳到另一个函数，然后又跳到下一个函数，对执行的逻辑难以理解。跳跃超过三次，基本就难以继续，只能放弃。第二个问题和第一个问题强相关。因为了解不够系统，很难形成整体的内核执行逻辑。而实际工作中碰到的问题总是千变万化，个人了解的一块未必能碰到。比如一个文件系统只读问题，是内核 VFS 层的问题？是文件系统自身？还是块设备或者硬盘的问题？如果不能形成清晰的视图，就很难有针对性的调试和改进。

按照方法论的观点，通常人类的学习过程是从易到难、从部分到整体、从已知到未知。而对内核的学习有其特殊之处，内核几乎是九十度的学习曲线，极难找到入门的路径，更别说快速流畅地阅读内核代码了。从那时起，笔者开始对内核进行整理，希望能找到一条学习的路径，在不断探索过程中，逐渐形成一份文档，然后通过一些培训活动验证了其有效，最终形成了本书。

本书可以归纳为两个思路。一个是对内核代码的分类。笔者把内核分为基础部分和应用部分。内核中的内存管理、任务调度和中断异常处理归为基础部分。而文件系统、设备管理和驱动归为应用部分。打开一份完整的内核代码统计一下，应用部分占了绝大多数，庞大复杂，但冗余很多，很多代码具有相似性；而基础部分则是短小精悍。应用部分经常要调用基础部分提供的内存管理、任务调度等服务。为了快速理解基础部分，首先要整理基础部分的服务，理解在内核中如何使用基础部分的服务。

第二个思路是把内核分为内核架构和内核实现。内核架构是内核中通用的、具普遍性的层次，比如块设备、字符设备、总线、文件系统的 VFS 层等。理解了内核架构，就对内核有了整体上的掌握，就能了解内核设计者的思路，进而快速流畅地阅读内核代码。但即使理解了内核架构，也还有很多具体问题要攻克。比如驱动中一个寄存器的使用、设备链路状态如何检测、文件系统如何使用 barrier I/O、同步和异步 I/O 的区别等。这是需要开发人员仔细研读和琢磨的。本书试图归纳整理出内核的常用架构层，这些架构层具有举一反三的作用，它们构成了 Linux 内核的骨架。

发展到今天，内核已经非常庞大和复杂。本书希望通过一些架构层次代码的分析，结合简单的例子，帮助读者理解内核的整体框架。当碰到内核问题或者需要加入某些内核功能或者修改某些实现时，可以迅速流畅地阅读相关代码，确定自己的方案，而不至于茫然无措。而对于细节的实现，则需要程序员根据自己的需求来设计。

关于内核版本，本书用的是 2.6.18 版。内核有一套自己的不兼容策略，不同内核版本之间经常不能编译，至于函数消失和数据结构修改更是家常便饭。所以我们只能选择一个版本作为基础。

阅读内核代码前的准备：下载一份完整的内核，Linux 内核的官方网站是 http://www.kernel.org，这里可以下载到各个版本的内核；再准备一个好的代码阅读软件，因为内核代码经常要前后关联阅读，所以需要具有代码工程管理的软件，强烈推荐 source insight，这是国内应用很广的一个软件。

另外，本书已经假设读者能编译和安装模块，并且具有计算机基本结构的知识。此外，有一台已安装了 Linux 系统的计算机或者虚拟机，并且经常实战练习。

由于笔者水平有限，而且从架构层次分析内核代码，可用来参考的资料很少，希望广大读者能多提意见，共同推进中国的技术水平。

任何书籍都不能替代读者自己对内核实际代码的研究和学习。但如果没有书籍，浩如云烟的内核代码让有志学习者茫然，而低效率地一点点啃代码也会浪费大量的时间。书籍的作用是带领读者入门，读者需要尽快转入自我学习阶段，对需要的部分代码自行分析和研究。

读者对象

本书适合以下读者阅读参考：

- ❑大专院校在校学生；
- ❑对系统内核感兴趣，有志于从事内核研发的人员；
- ❑从事驱动开发的工程师；
- ❑从事操作系统内核方面工作的工程师；
- ❑负责 Linux 系统调试和优化的技术人员。

如何阅读本书

本书将整个内核分为基础层和应用层。这种划分大大减少了阅读内核的难度，但是仍然需要对基础层有全面正确的理解。本书第 1 章介绍了内核的基础层，读者应该多做一些实践练习，才能加深理解。

第 2 章是本书提纲挈领的一章，重点介绍了文件系统的基础知识。文件系统在应用层的位置非常重要，因此只有掌握了文件系统的重要概念、理解了基本的操作过程才能为整体理解内核打下良好的基础。

第 3 ～ 9 章是关于设备的章节。建议读者结合具体的设备，从设备到总线再到驱动，逐步深入。本书的章节安排遵循从易到难、代码结合实例的方式，相信读者可以比较顺畅地阅读并理解。

第 10 ～ 13 章，再次对文件系统的读写和内核通用块层进行阐述。阅读的过程中，读者若能结合实际做一些小的程序，则可以帮助迅速提升能力。比如自己实现内核的 I/O 路径或者实现一个模拟的块设备系统，实践中应用才是能力提升的最佳途径。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。书中的全部源文件可以从华章网站⊖下载。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 easyblue99@hotmail.com，期待能够得到你们的真挚反馈。

致谢

感谢 IBM 公司的小强、中兴的老谢、百度的子旬、索尼的大 A 以及淘宝、Intel、微软、搜狐、

⊖ 参见华章网站 www.hzbook.com。——编辑注

新浪等公司的各位朋友。与你们的一次次讨论澄清了很多概念和问题，使我受益良多。

感谢机械工业出版社华章公司的编辑白宇，是你认真审核每一章的内容，提高了书稿的质量。编辑是个清苦的职业，远远谈不上令人羡慕，但不论多少生活的烦恼，白女士总是耐心修改，职业精神令人敬佩。

感谢我的亲人们，特别要感谢我的母亲，她是中国核工业的早期建设者，在艰难中养育了三个孩子，谨以此书献给我的母亲杨玉芳女士。感谢我的妻子和女儿，你们是我永远的动力之源。

目 录

第 1 章

内核的基础层和应用层

前言中提到，内核分为内核基础层和内核应用层。这既有对整个操作系统软件架构的分析和理解，也有现实应用情况的支持。

操作系统对应用软件提供了统一的编程接口，操作系统的系统调用是稳定的、向下兼容的，但是在内核中，并不提供这种稳定且兼容的保证。实际上，同样的代码在不同的内核版本经常可能编译失败。内核的这种开发模式，造成了学习内核时版本众多而且不稳定的特点，也大大增加了学习的困难。

在长期对内核代码的分析和应用中，笔者注意到一个事实：内核中提供了大量的软件基础设施。这些基础设施既包括内核中对内存的使用，对进程调度的控制，也包括自旋锁、信号量等内核提供的同步函数，同时还包括内核提供的数据结构，比如链表、hash 链表、红黑树等。这些软件基础设施如同操作系统提供的系统调用一样，是理解内核代码和编写内核代码的基础。而这些软件基础设施在各个内核版本中基本是稳定的。

现实情况提供了另一方面的支持。学习的动力来自于应用，传统的操作系统教科书全面，但也很少有人能完全读懂并且结合代码进行实战应用。大多数程序员在工作中应用到内核的部分，绝大多数是设备驱动，而讲操作系统的书多数不会关注到设备驱动层面。除了设备驱动之外，内核中文件系统也有较多的应用。

要做到快速流畅地阅读内核代码，前提是了解内核中的软件基础设施。这些知识使用范围很广，分布在内核代码的各个部分，如果不了解，在内核代码的理解上就容易出现障碍。

1.1 内核基础层提供的服务

操作系统通常提供的服务是内存管理、进程管理、设备管理和文件系统。本书将内存管理、进程管理以及其他内核提供的基础软件设施，比如工作队列、tasklet 以及信号量和自旋锁都作为内核的基础层。本书并不分析和探讨这些基础层的原理和实现，本章只介绍如何使用这些基础软件设施。

1.1.1 内核中使用内存

简单说，内核提供了两个层次的内存分配接口。一个是从伙伴系统分配，另一个是从 slab 系统分配。伙伴系统是最底层的内存管理机制，提供页式的内存管理，而 slab 是伙伴系统之上的内存管理，提供基于对象的内存管理。

从伙伴系统分配内存的调用是 alloc_pages，注意此时得到的是页面地址，如果要获得能使用的内存地址，还需要用 page_address 调用来获得内存地址。

如果要直接获得内存地址，需要使用 __get_free_pages。__get_free_pages 其实封装了 alloc_pages 和 page_address 两个函数。

alloc_pages 申请的内存是以页为单元的，最少要一个页。如果只是申请一小块内存，一个页就浪费了，而且内核中很多应用也希望一种对象化的内存管理，希望内存管理能自动地构造和析构对象，这都很接近面向对象的思路了，这就是 slab 内存管理。

要从 slab 申请内存，需要创建一个 slab 对象，使用 kmem_cache_create 创建 slab 对象。kmem_cache_create 可以提供对象的名字和大小、构造函数和析构函数等，然后通过 kmem_cache_alloc 和 kmem_cache_free 来申请和释放内存。

内核中常用的 kmalloc 其实也是 slab 提供的对象管理，只不过内核已经构建了一些固定大小的对象，用户通过 kmalloc 申请的时候，就使用了这些对象。

一个内核中创建 slab 对象的例子如代码清单 1-1 所示。

代码清单 1-1 创建 slab 对象

```
bh_cachep = kmem_cache_create("buffer_head",
                    sizeof(struct buffer_head), 0,
                    (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD),
                    init_buffer_head,
                    NULL);
```

创建一个 slab 对象时指定了 slab 对象的大小，用以下代码申请一个 slab 对象：

```
struct buffer_head *ret = kmem_cache_alloc(bh_cachep, gfp_flags);
```

内核中还有一个内存分配调用：vmalloc。vmalloc 的作用是把物理地址不连续的内存页面拼凑为逻辑地址连续的内存区间。

理解了以上几个函数调用之后，阅读内核代码的时候就可以清晰内核中对内存的使用方式。

1.1.2 内核中的任务调度

内核中经常需要进行进程的调度。首先看一个例子，如代码清单 1-2 所示。

代码清单 1-2 使用 wait 的任务调度

```
#define wait_event(wq, condition)
do {
    if (condition)
```

```
        break;
    __wait_event(wq, condition);
} while (0)

#define __wait_event(wq, condition)
do {
    DEFINE_WAIT(__wait);

    for (;;) {
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
        if (condition)
            break;
        schedule();
    }
    finish_wait(&wq, &__wait);
} while (0)
```

上文定义了一个 wait 结构，然后设置进程睡眠。如果有其他进程唤醒这个进程后，判断条件是否满足，如果满足，删除 wait 对象，否则进程继续睡眠。

这是一个很常见的例子，使用 wait_event 实现进程调度的实例在内核中很多，而且内核中还实现了一系列函数，简单介绍如下。

- wait_event_timeout：和 wait_event 的区别是有时间限制，如果条件满足，进程恢复运行，或者时间到达，进程同样恢复运行。
- wait_event_interruptible：和 wait_event 类似，不同之处是进程处于可中断的睡眠。而 wait_event 设置进程处于不可中断的睡眠。两者区别何在？可中断的睡眠进程可以接收到信号，而不可中断的睡眠进程不能接收信号。
- wait_event_interruptible_timeout：和 wait_event_interruptible 相比，多个时间限制。在规定的时间到达后，进程恢复运行。
- wait_event_interruptible_exclusive：和 wait_event_interruptible 区别是排他性的等待。



注意

何谓排他性的等待？有一些进程都在等待队列中，当唤醒的时候，内核是唤醒所有的进程。如果进程设置了排他性等待的标志，唤醒所有非排他性的进程和一个排他性进程。

1.1.3 软中断和 tasklet

Linux 内核把对应中断的软件执行代码分拆成两部分。一部分代码和硬件关系紧密，这部分代码必须关闭中断来执行，以免被后面触发的中断打断，影响代码的正确执行，这部分代码放在中断上下文中执行。另一部分代码和硬件关系不紧密，可以打开中断执行，这部分代码放在软中断上下文执行。

需要指出的是，这种划分是一种粗略、大概的划分。中断是计算机系统的宝贵资源，关闭中断意味着系统不响应中断，这常常是代价高昂的。所以为了避免关闭中断的不利影响，

即使在中断上下文中，也有很多代码的执行是打开中断的。而在软中断上下文，甚至进程上下文的内核代码中，有的时候也是需要关闭中断的。哪些地方需要关闭中断，而哪些地方又可以打开中断，需要仔细地考虑，既要尽可能打开中断以防止关闭中断的不利影响，又要在需要的时候关闭中断以避免出现错误。

Linux 内核定义了几个默认的软中断，网络设备有自己的发送和接收软中断，块设备也有自己的软中断。为了方便使用，内核还定义了一个 tasklet 软中断。tasklet 是一种特殊的软中断，同一时刻一个 tasklet 只能有一个 CPU 执行，不同的 tasklet 可以在不同的 CPU 上执行。这和软中断不同，软中断同一时刻可以在不同的 CPU 并行执行，因此软中断必须考虑重入的问题。

内核中很多地方使用了 tasklet。分析一个例子，代码如下所示：

```
DECLARE_TASKLET_DISABLED(hil_mlcs_tasklet, hil_mlcs_process, 0);
tasklet_schedule(&hil_mlcs_tasklet);
```

上面的例子首先定义了一个 tasklet，它的执行函数是 hil_mlcs_process。当程序中调用 tasklet_schedule，会把要执行的结构插入到一个 tasklet 链表，然后触发一个 tasklet 软中断。每个 CPU 都有自己的 tasklet 链表，内核会根据情况确定在何时执行 tasklet。

可以看到，tasklet 使用起来很简单，本节只需要了解在内核如何使用即可。

1.1.4 工作队列

工作队列和 tasklet 相似，都是一种延缓执行的机制。不同之处是工作队列有自己的进程上下文，所以工作队列可以睡眠，也可以被调度，而 tasklet 不可睡眠。代码清单 1-3 是工作队列的例子。

代码清单 1-3 工作队列

```
INIT_WORK(&ioc->sas_persist_task,
   mptsas_persist_clear_table,
   (void *)ioc);
schedule_work(&ioc->sas_persist_task);
```

使用工作队列很简单，schedule_work 把用户定义的 work_struct 加入系统的队列中，并唤醒系统线程去执行。那么是哪个系统线程执行用户的 work_struct 呢？实际上，内核初始化的时候，就要创建一个工作队列 keventd_wq，同时为这个工作队列创建内核线程（默认是为每个 CPU 创建一个内核线程）。

内核同时还提供了 create_workqueue 和 create_singlethread_workqueue 函数，这样用户可以创建自己的工作队列和执行线程，而不用内核提供的工作队列。看内核的例子：

```
kblockd_workqueue = create_workqueue("kblockd");
int kblockd_schedule_work(struct work_struct *work){
    return queue_work(kblockd_workqueue, work);
}
```

kblockd_workqueue 是内核通用块层提供的工作队列，需要由 kblockd_workqueue 执行的工作就要调用 kblockd_schedule_work，其实就是调用 queue_work 把 work 插入到

kblockd_workqueued 的任务链表。

create_singlethread_workqueue 和 create_workqueue 类似，不同之处是，像名字揭示的一样，create_singlethread_workqueue 只创建一个内核线程，而不是为每个 CPU 创建一个内核线程。

1.1.5 自旋锁

自旋锁用来在多处理器的环境下保护数据。如果内核发现数据未锁，就获取锁并运行；如果数据被锁，就一直旋转（其实是一直反复执行一条指令）。之所以说自旋锁用在多处理器环境，是因为在单处理器环境（非抢占式内核）下，自旋锁其实不起作用。在单处理器抢占式内核的情况下，自旋锁起到禁止抢占的作用。

因为被自旋锁锁着的进程一直旋转，而不是睡眠，所以自旋锁可以用在中断等禁止睡眠的场景。自旋锁的使用很简单，请参考下面的代码例子。

```
spin_lock(shost->host_lock);
shost->host_busy++;
spin_unlock(shost->host_lock);
```

1.1.6 内核信号量

内核信号量和自旋锁类似，作用也是保护数据。不同之处是，进程获取内核信号量的时候，如果不能获取，则进程进入睡眠状态。参考代码如下：

```
down(&dev->sem);
up(&dev->sem);
```

内核信号量和自旋锁很容易混淆，所以列出两者的不同之处。

- 内核信号量不能用在中断处理函数和 tasklet 等不可睡眠的场景。
- 深层次的原因是 Linux 内核以进程为单位调度，如果在中断上下文睡眠，中断将不能被正确处理。
- 可睡眠的场景既可使用内核信号量，也可使用自旋锁。自旋锁通常用在轻量级的锁场景。即锁的时间很短，马上就释放锁的场景。

1.1.7 原子变量

原子变量提供了一种原子的、不可中断的操作。如下所示：

```
atomic_t            mapped;
```

内核提供了一系列的原子变量操作函数，如下所示。

- atomic_add：加一个整数到原子变量。
- atomic_sub：从原子变量减一个整数。
- atomic_set：设置原子变量的数值。
- atomic_read：读原子变量的数值。

1.2 内核基础层的数据结构

内核使用的数据结构有双向链表、hash 链表和单向链表，另外，红黑树和基树（radix 树）也是内核使用的数据结构。实际上，这也是程序代码中通常使用的数据结构。

container 是 Linux 中很重要的一个概念，使用 container 能实现对象的封装。代码如下所示：

```
#define container_of(ptr, type, member) ({
     const typeof( ((type *)0)->member ) *__mptr = (ptr);
     (type *)( (char *)__mptr - offsetof(type,member) );})
```

这个方法巧妙地实现了通过结构的一个成员找到整个结构的地址。内核中大量使用了这个方法。

1.2.1 双向链表

list 是双向链表的一个抽象，它定义在 /include/linux 目录下。首先看看 list 的结构定义：

```
struct list_head {
struct list_head *next, *prev;
};
```

list 库提供的 list_entry 使用了 container，通过 container 可以从 list 找到整个数据对象，这样 list 就成为了一种通用的数据结构：

```
#define list_entry(ptr, type, member)
 container_of(ptr, type, member)
```

内核定义了很多对 list 结构操作的内联函数和宏。

- LIST_HEAD：定义并初始化一个 list 链表。
- list_add_tail：加一个成员到链表尾。
- list_del：删除一个 list 成员。
- list_empty：检查链表是否为空。
- list_for_each：遍历链表。
- list_for_each_safe：遍历链表，和 list_for_each 的区别是可以删除遍历的成员。
- list_for_each_entry：遍历链表，通过 container 方法返回结构指针。

1.2.2 hash 链表

hash 链表和双向链表 list 很相似，它适用于 hash 表。看一下 hash 链表的头部定义 :

```
struct hlist_head {
struct hlist_node *first;
};
```

和通常的 list 比较，hlist 只有一个指针，这样就节省了一个指针的内存。如果 hash 表非常庞大，每个 hash 表头节省一个指针，整个 hash 表节省的内存就很可观了。这就是内核中专门定义 hash list 的原因。

hash list 库提供的函数和 list 相似，具体如下。

- HLIST_HEAD：定义并初始化一个 hash list 链表头。
- hlist_add_head：加一个成员到 hash 链表头。
- hlist_del：删除一个 hash 链表成员。
- hlist_empty：检查 hash 链表是否为空。
- hlist_for_each：遍历 hash 链表。
- hlist_for_each_safe：遍历 hash 链表，和 hlist_for_each 的区别是可以删除遍历的成员。
- hlist_for_each_entry：遍历 hash 链表，通过 container 方法返回结构指针。

1.2.3 单向链表

内核中没有提供单向链表的定义。但实际上，有多个地方使用了单向链表的概念，看代码清单 1-4 的例子。

代码清单 1-4 单向链表

```
for (i = 0, p -= n; i < n; i++, p++, index++) {
     struct probe **s = &domain->probes[index % 255];
     while (*s && (*s)->range < range)
          s = &(*s)->next;
     p->next = *s;
     *s = p;
}
```

上面的例子是字符设备的 map 表，probe 结构其实就是单向链表。这种结构在内核中应用很广泛。

1.2.4 红黑树

红黑树是一种自平衡的二叉树，代码位于 /lib/rbtree.c 文件。实际上，红黑树可以看做折半查找。我们知道，排序的快速做法是取队列的中间值比较，然后在剩下的队列中再次取中间值比较，迭代进行，直到找到最合适的数据。红黑树中的“红黑”代表什么意思呢？红黑的颜色处理是避免树的不平衡。举个例子，如果 1、2、3、4、5 五个数字依次插入一颗红黑树，那么五个值都落在树的右侧，如果再将 6 插入这颗红黑树，要比较五次。为避免这种情况，“红黑规则”就要将树旋转，树的根部要落在 3 这个节点，这样就避免了树的不平衡导致的问题。

内核中的 I/O 调度算法和内存管理中都使用了红黑树。红黑树也有很多介绍的文章，读者可以结合代码分析一下。

1.2.5 radix 树

内核提供了一个 radix 树库，代码在 /lib/radix-tree.c 文件。radix 树是一种空间换时间的数据结构，通过空间的冗余减少了时间上的消耗。radix 树的形象图如图 1-1 所示。

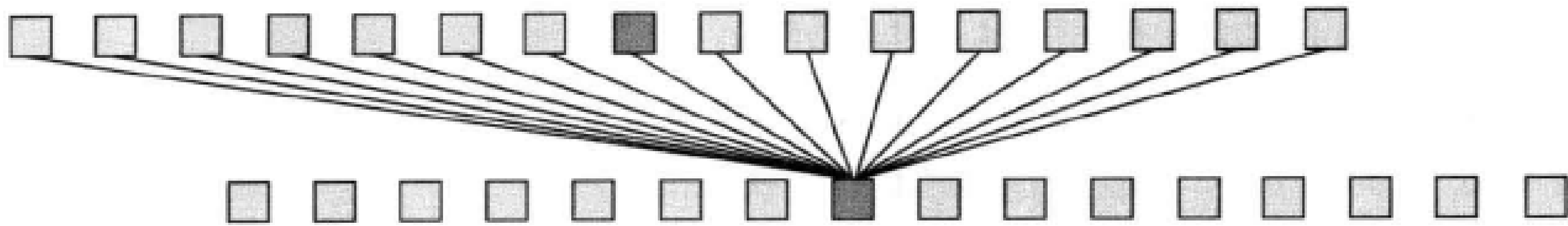


图 1-1 radix 树的形象图

如图 1-1 所示，元素空间总数为 256，但元素个数不固定。如果用数组存储，好处是插入查找只用一次操作，但是存储空间需要 256，这是不可思议的。如果用链表存储，存储空间节省了，但是极限情况下查找元素的次数等于元素的个数。而采用一颗高度为 2 的基树，第一级最多 16 个冗余成员，代表元素前四位的索引，第二级代表元素后四位的索引。只要两级查找就可以找到特定的元素，而且只有少量的冗余数据。图中假设只有一个元素 10001000，那么只有树的第一级有元素，而且树的第二级只有 1000 这个节点有数据，其他节点都不必分配空间。这样既可以快速定位查找，也减少了冗余数据。

radix 树很适合稀疏的数据，内核中文件的页缓存就采用了 radix 树。关于 radix 树的文章很多，读者可以结合内核 radix 树的实现代码分析一下。

1.3 内核应用层

内核应用层是建立在基础层之上的功能性系统。在本书中，内核应用层指的是文件系统、设备、驱动以及网络。内核代码虽然庞杂，但是核心的基础层并不庞大，主要是应用层占据了大部分代码量。图 1-2 展示了内核各部分的代码量统计数据。

TYPE	COUNT	PER CENT
Drivers	3,301,081	51.6
Architectures	1,258,638	19.7
Filesystems	544,871	8.5
Networking	376,716	5.9
Sound	356,180	5.6
Include	320,078	5.0
Kernel	74,503	1.2
Memory Mgmt	36,312	0.6
Cryptography	32,769	0.5
Security	25,303	0.4
Other	72,780	1.1

图 1-2 内核代码的统计数据

从图 1-2 可以计算得出，驱动、文件系统和网络占据了内核代码的绝大部分，而代表基础层的 kernel 和内存管理实际上只有很少的代码量。Architectures 属于内核的基础层，它是为适配不同的 CPU 结构提供了不同的代码，对某种 CPU 来说（如读者最关注的 x86CPU），实际的代码量也大大减少了。

1.4 从 Linux 内核源码结构纵览内核

本节通过 Linux 内核源码的各个目录来分析内核代码的数量和阅读难度。如图 1-3 所示。

从图 1-3 可以发现，Architectures 的子目录是各个 CPU 架构的名字，为各种不同的 CPU 架构服务。虽然总体量很大，但是对读者关注的 x86 或者 ARM 来说，也只占很小的一部分。

图 1-4 展示了内核中 drivers 目录的分类。

drivers 目录分类为各种不同的设备驱动，而设备驱动虽然五花八门，但是它们的结构是高度相似的，读者可以根据工作需要阅读分析驱动代码。在理解设备驱动架构的基础上，这个工作具有高度重复性，可以在短时间内掌握驱动的精髓。

图 1-5 展示了内核中 fs 目录的分类。fs 目录分类为各种不同的文件系统。

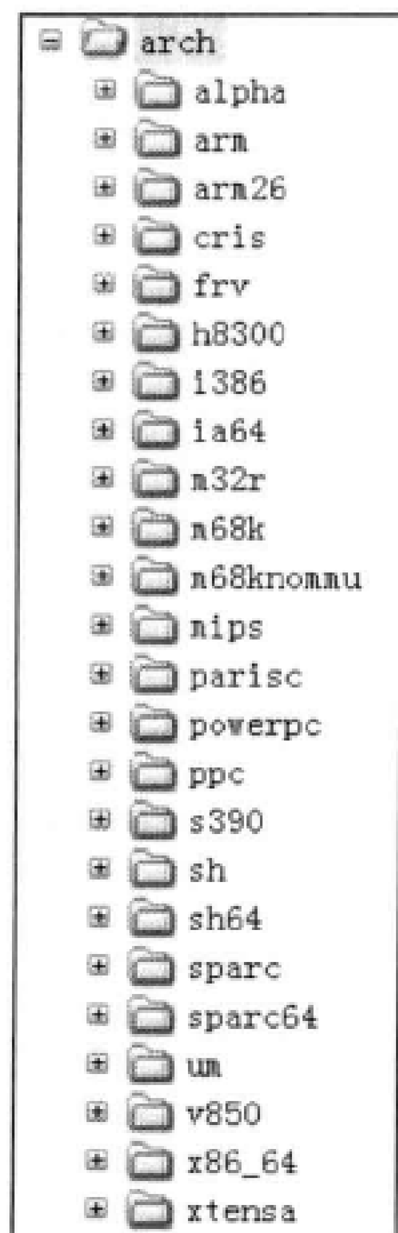


图 1-3 内核中 Architectures 目录的分类

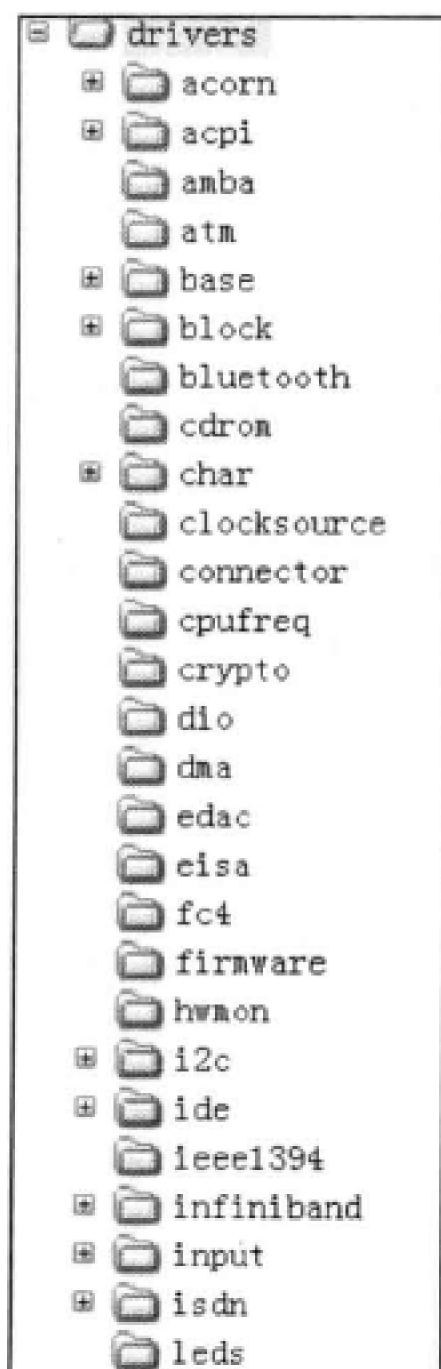


图 1-4 内核中 drivers 目录的分类

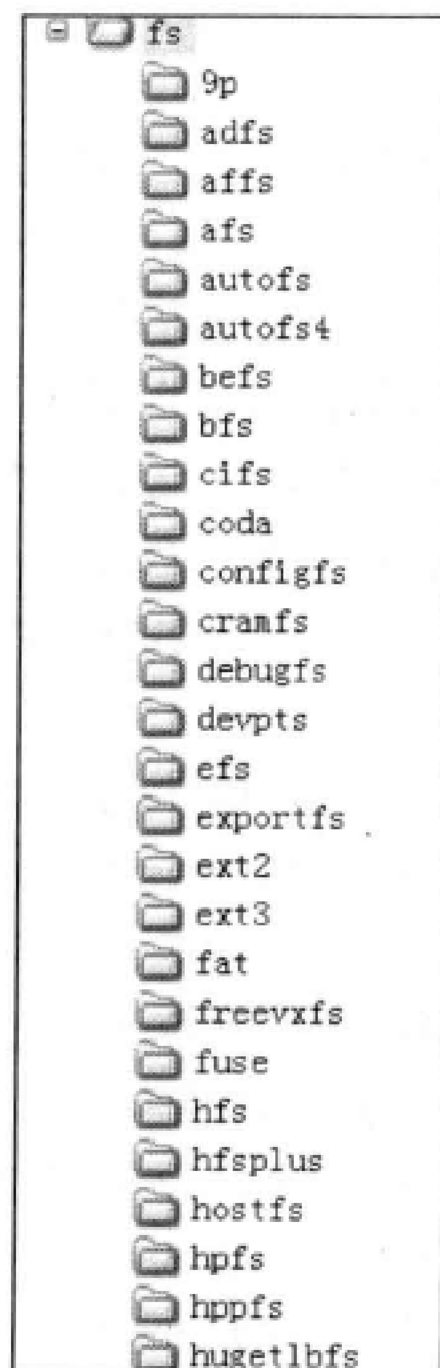


图 1-5 内核中 fs 目录的分类

Linux 为文件系统统一提供了一个 VFS 架构，各种文件系统都要按照这个架构来设计。

因此，各种不同的文件系统都具有重复的部分，读者不需要逐一分析所有的文件系统代码，只选择几种文件系统重点阅读即可。

1.5 内核学习和应用的四个阶段

如何深入学习内核？或者更进一步，如果把内核知识应用到具体的工作中，对工作产生价值？

根据所有内核核心开发人员的观点，阅读代码、理解代码是最重要的步骤。对于操作系统这种庞大复杂的基础软件来说，只学习操作系统教程之类的书籍，远远不能达到理解和应用的目的。这是所有工程实践类学科的特点，“只在岸上学习是永远不可能掌握游泳技巧的”。所以必须以代码为依托，以代码为依归。

其次，是如何选择代码的问题。内核代码非常庞大，如何在开始阶段选择代码，既能覆盖主要方面，同时又不至于难度太高，是分析和学习的主要问题。本书把内核代码分为了基础层和应用层，对基础层突出内核 API 的概念，在应用层的分析中，希望通过突出重点架构和选择典型例子来加深理解。为了便于阅读，笔者将简单的代码注释直接加在代码清单里，需要重点解释的部分，在正文加以说明。

学以致用，任何学习的坚实基础都不只是单纯的兴趣，而是要在实践中得到检验。所以检验学习效果，要看实际的应用，而不能只是单方面地阅读分析代码。实际的应用和学习过程，笔者认为，可以粗略地分为四个阶段。

（1）起步阶段

结合中国当前的应用现状，起步阶段基本都是从驱动入手。这一阶段的表现是，实际做过几个驱动，能够移植驱动到不同的系统平台，对驱动能够做一定的修改，能够裁剪内核，以适应具体的需求；对 Linux 的 bootloader 能够根据需求做修改。根据笔者对国内现状的了解和调查，大多数国内的内核应用停留在这个层面，大多数内核相关的工作也是在这个层面进行。

（2）熟练阶段

对内核的一个或者几个部分比较熟悉，针对熟悉部分，可以进行深度的开发应用。比如对设备驱动相关的总线、设备、中断比较熟悉，并且可以做深层次的开发。这一阶段的特点是对内核的理解还不够全面，需要时间积累增加对内核整体的把握。

（3）高级阶段

对整个内核的重要部分都进行了比较深入的分析。这一阶段的特点是全面性，即使要学习内核某些新的重要特性，也能在短时间内迅速掌握重点。

（4）终极阶段

此阶段是 Linux 内核组维护人员所达到的水准，能做开创性的工作，具有重大的应用价值。处于这个阶段的主要是欧美的资深开发人员（或者说是内核 hacker），国内达到这个水准的技术人员非常少。

1.6 本章小结

内核基础层是整个内核学习的基础。基础层的实现部分是比较复杂的，尤其是内存管理基础层和进程调度基础层。但是应用这些基础层并不复杂，它们的应用接口 API 也比较稳定，在各个内核版本中没有太大的变化，在内核代码中经常会调用这些基础层的接口 API。读者在阅读内核代码的时候，可以回顾这部分的知识，增强熟练运用的能力。

第 2 章

文件系统

本章从文件系统开始探索整个内核应用层。在本书的整个体系中，文件系统担当着最重要的作用，可以认为，Linux 内核的应用层就是以文件系统为核心而展开的。

本书以文件系统作为整个内核应用层的核心，主要基于下列理由。

- **文件系统本身具有重大作用**。国内一些技术公司，已经开始对文件系统的深度研究和应用。这其中，既有通信公司，也有传统 IT 公司和互联网公司。当前，分布式文件系统的广泛应用让文件系统成为当前内核应用的热门。
- **文件系统在整个内核架构中具有基础架构性质**。字符设备、块设备这些设备驱动的概念都要依靠文件系统来实现。设备管理的基础架构也要依赖文件系统（sysfs）。而设备和驱动是国内当前在内核层面应用最多的方面，也是国内程序员在底层开发中应用最多的方面。

前言讲到，书籍的作用只是带领读者入门，带领入门的关键是提供一条学习分析的快捷路径。根据笔者的经验，从架构层面分析内核，难点就是如何划分代码的层次。如何找到一个核心点，然后从核心点扩展，每一部分都自成单元，既具有普遍性，又能向外扩展，是本文最大一个挑战。如果没有建立层次分明的架构体系，内核代码就会显得庞大混乱，难以梳理和学习。从文件系统入手，掌握基本概念和实现架构后，可以从文件系统引出设备文件的概念，设备文件又可以引申到字符设备和块设备，这样就从文件系统过渡到设备管理。设备管理包含了设备驱动，设备驱动要用到中断，设备里面的块设备又控制了通用块层和 I/O 调度。而文件系统向外引申又和网络的 socket 联系。深入文件系统的代码，可以了解到内存的页面管理。从文件系统出发，层次推进基本囊括了内核应用层的重要概念和架构。这是作者总结的一种学习体系，希望通过这种体系串联起内核的知识点。

2.1 文件系统的基本概念

在深入分析文件系统之前，有必要介绍文件系统的几个基本概念，这将从架构层次理解文件系统设计的目的，从而从全局层面理解内核文件系统的代码，大大减低分析代码的

难度和工作量。

2.1.1 什么是 VFS

Linux 内核通过虚拟文件系统（Virtual File System，VFS）管理文件系统。

VFS 是 Linux 内核文件系统的一个极其重要的基础设施，VFS 为所有的文件系统提供了统一的接口，对每个具体文件系统的访问要通过 VFS 定义的接口来实现。同时 VFS 也是一个极其重要的架构，所有 Linux 的文件系统必须按照 VFS 定义的方式来实现。

VFS 本身只存在于内存中，它需要将硬盘上的文件系统抽象到内存中，这个工作是通过几个重要的结构实现的。VFS 定义了几个重要的结构：dentry、inode、super_block，通过这些结构将一个真实的硬盘文件系统抽象到了内存，从而通过管理 dentry、inode 这几个对象就可以完成对文件系统的一些操作（当然，在合适的时候，仍然需要将内存的数据写入到硬盘）。

2.1.2 超级块 super_block

超级块（super_block）代表了整个文件系统本身。通常，超级块是对应文件系统自身的控制块结构（可参考 ext2 文件系统的超级块结构）。超级块保存了文件系统设定的文件块大小，超级块的操作函数，而文件系统内所有的 inode 也都要链接到超级块的链表头。对于一个具体文件系统的控制块可能还含有另外的信息，而通过超级块对象，我们可以找到这些必要的信息。

超级块的内容需要读取具体文件系统在硬盘上的超级块结构获得，所以超级块是具体文件系统超级块的内存抽象。超级块对象整个结构很庞大复杂，作为学习的起步阶段，没必要探究每个成员的具体意义和使用目的（实际上，强记也很容易忘记）。代码清单 2-1 是超级块对象简化后的结构定义。

代码清单 2-1 超级块结构简化定义

```
struct super_block {
 unsigned long             s_blocksize;
 unsigned char             s_blocksize_bits;
 ....../* 省略超级块的链表、设备号等代码 */
 unsigned long long        s_maxbytes;   /* Max file size */

 struct file_system_type   *s_type;
 struct super_operations   *s_op;

 unsigned long             s_magic;
 struct dentry             *s_root;

 struct list_head          s_inodes;     /* all inodes */
 struct list_head          s_dirty;      /* dirty inodes */

 struct block_device       *s_bdev;
```

```
    void                        *s_fs_info;   /* Filesystem private info */
};
```

从两个方面了解超级块结构的作用。

1）超级块结构给出了文件系统的全局信息。

- s_blocksize 指定了文件系统的块大小。
- s_maxbytes 指定文件系统中最大文件的尺寸。
- s_type 是指向 file_system_type 结构的指针。
- s_magic 是魔术数字，每个文件系统都有一个魔术数字。
- s_root 是指向文件系统根 dentry 的指针。

超级块对象还定义了一些链表头，用来链接文件系统内的重要成员。

- s_inodes 指向文件系统内所有的 inode，通过它可以遍历 inode 对象。
- s_dirty 指向所有 dirty 的 inode 对象。
- s_bdev 指向文件系统存在的块设备指针。

在后面具体文件系统的例子中，可以看到这些成员是如何赋值和应用的。

2）超级块结构包含一些函数指针。

super_operations 提供了最重要的超级块操作。例如 super_operation 的成员函数 read_inode 提供了读取 inode 信息的功能。每个具体的文件系统一般都要提供这个函数来实现对 inode 信息的读取，例如 ext2 文件系统提供的具体函数是 ext2_read_inode。从这个例子，我们可以理解“VFS 提供了架构，而具体文件系统必须按照 VFS 的架构去实现”的含义。

2.1.3 目录项 dentry

对于一个通常的文件系统来说，文件和目录一般按树状结构保存。直观来看，目录里保存着文件，而所有目录一层层汇聚，最终到达根目录。从这个树状结构，我们可以想象 VFS 应该有数据结构反映这种树状的结构。实际上确实如此，目录项（dentry）就是反映了文件系统的这种树状关系。

在 VFS 里，目录本身也是一个文件，只是有点特殊。每个文件都有一个 dentry（可能不止一个），这个 dentry 链接到上级目录的 dentry。根目录有一个 dentry 结构，而根目录里的文件和目录都链接到这个根 dentry，二级目录里的文件和目录，同样通过 dentry 链接到二级目录。这样一层层链接，就形成了一颗 dentry 树。从树顶可以遍历整个文件系统的所有目录和文件。

为了加快对 dentry 的查找，内核使用了 hash 表来缓存 dentry，称为 dentry cache。dentry cache 在后面的分析中经常用到，因为 dentry 的查找一般都先在 dentry cache 里进行查找。

dentry 的结构定义同样很庞杂，和超级块类似，我们当前只分析最重要的部分。dentry 结构简化后的定义如代码清单 2-2 所示。

代码清单 2-2 dentry 结构的简化定义

```
struct dentry {
```

```
    ……/ 省略 dentry 锁、标志等代码 /
    struct inode *d_inode;   /* Where the name belongs to - NULL is negative */
    /*
     * The next three fields are touched by __d_lookup.  Place them here
     * so they all fit in a cache line.
     */
    struct hlist_node d_hash;       /* lookup hash list */
    struct dentry *d_parent;        /* parent directory */
    struct qstr d_name;

    /*
     * d_child and d_rcu can share memory
     */
    union {
      struct list_head d_child;     /* child of parent list */
      struct rcu_head d_rcu;
    } d_u;
    struct list_head d_subdirs;     /* our children */

    struct dentry_operations *d_op;
    struct super_block *d_sb;       /* The root of the dentry tree */
    int d_mounted;
   };
```

对 dentry 结构的成员解释如下。

- d_inode 指向一个 inode 结构。这个 inode 和 dentry 共同描述了一个普通文件或者目录文件。
- dentry 结构里有 d_subdirs 成员和 d_child 成员。d_subdirs 是子项（子项可能是目录，也可能是文件）的链表头，所有的子项都要链接到这个链表。d_child 是 dentry 自身的链表头，需要链接到父 dentry 的 d_subdirs 成员。当移动文件的时候，需要把一个 dentry 结构从旧的父 dentry 的链表上脱离，然后链接到新的父 dentry 的 d_subdirs 成员。这样 dentry 结构之间就构成了一颗目录树
- d_parent 是指针，指向父 dentry 结构。
- d_hash 是链接到 dentry cache 的 hash 链表。
- d_name 成员保存的是文件或者目录的名字。打开一个文件的时候，根据这个成员和用户输入的名字比较来搜寻目标文件。
- d_mounted 用来指示 dentry 是否是一个挂载点。如果是挂载点，该成员不为零。

2.1.4 索引节点 inode

inode 代表一个文件。inode 保存了文件的大小、创建时间、文件的块大小等参数，以及对文件的读写函数、文件的读写缓存等信息。一个真实的文件可以有多个 dentry，因为指向文件的路径可以有多个（考虑文件的链接），而 inode 只有一个。

根据上面的描述是否可以得出结论，即 dentry 和 inode 代表一个文件？事实基本如此，

inode 和 dentry 分别代表了文件通用的两个部分，只不过对某些文件系统而言，inode 提供的信息还不够，还需要其他信息。这个将在后面具体的文件系统里面看到。

inode 的结构定义如代码清单 2-3 所示。因为 inode 的定义非常庞大，在我们初次认识 inode 结构的时候，将 inode 结构定义大大简化，以重点突出几个结构成员。

代码清单 2-3 inode 的结构定义

```
struct inode {
 struct list_head                i_list;
 struct list_head                i_sb_list;
 struct list_head                i_dentry;
 unsigned long                   i_ino;
 atomic_t                        i_count;
 loff_t                          i_size;

 unsigned int                    i_blkbits;
 struct inode_operations         *i_op;
 const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
 struct address_space            *i_mapping;
 struct block_device             *i_bdev;
  ……/* 省略锁等代码 */
};
```

inode 结构非常复杂，同样，我们只分析其中最重要的几个成员，以简单理解 inode 最重要的作用。而其他的成员在后面的章节中继续分析，逐渐丰富对 inode 的理解。

- 成员 i_list、i_sb_list、i_dentry 分别是三个链表头。成员 i_list 用于链接描述 inode 当前状态的链表。成员 i_sb_list 用于链接到超级块中的 inode 链表。当创建一个新的 inode 的时候，成员 i_list 要链接到 inode_in_use 这个链表，表示 inode 处于使用状态，同时成员 i_sb_list 也要链接到文件系统超级块的 s_inodes 链表头。由于一个文件可以对应多个 dentry，这些 dentry 都要链接到成员 i_dentry 这个链表头。
- 成员 i_ino 是 inode 的号，而 i_count 是 inode 的引用计数。成员 i_size 是以字节为单位的文件长度。
- 成员 i_blkbits 是文件块的位数。
- 成员 i_fop 是一个 struct file_operations 类型的指针。文件的读写函数和异步 io 函数都在这个结构中提供。每一个具体的文件系统，基本都要提供各自的文件操作函数。
- i_mapping 是一个重要的成员。这个结构目的是缓存文件的内容，对文件的读写操作首先要在 i_mapping 包含的缓存里寻找文件的内容。如果有缓存，对文件的读就可以直接从缓存中获得，而不用再去物理硬盘读取，从而大大加速了文件的读操作。写操作也要首先访问缓存，写入到文件的缓存。然后等待合适的机会，再从缓存写入硬盘。后面我们将分析文件的具体读写，在此处只需要理解基本的作用即可。
- 成员 i_bdev 是指向块设备的指针。这个块设备就是文件所在的文件系统所绑定的块设备。

2.1.5 文件

文件对象的作用是描述进程和文件交互的关系。这里需要指出的是，硬盘上并不存在一个文件结构。进程打开一个文件，内核就动态创建一个文件对象。同一个文件，在不同的进程中有不同的文件对象。

文件的结构定义如代码清单 2-4 所示。

代码清单 2-4　文件的数据结构

```
struct file {
 struct dentry                  *f_dentry;
 struct vfsmount                *f_vfsmnt;
 const struct file_operations   *f_op;
 ……/* 省略部分代码 */
 loff_t                         f_pos;
 struct fown_struct             f_owner;
 unsigned int                   f_uid, f_gid;
 struct file_ra_state           f_ra;

 struct address_space           *f_mapping;
};
```

- f_dentry 和 f_vfsmnt 分别指向文件对应的 dentry 结构和文件所属于的文件系统的 vfsmount 对象。
- 成员 f_pos 表示进程对文件操作的位置。例如对文件读取前 10 字节，f_pos 就指示第 11 字节位置。
- f_uid 和 f_gid 分别表示文件的用户 ID 和用户组 ID。
- 成员 f_ra 用于文件预读的设置。在第 10 章将继续详细分析预读的使用。
- f_mapping 指向一个 address_space 结构。这个结构封装了文件的读写缓存页面。

2.2 文件系统的架构

VFS 是具体文件系统的抽象，而 VFS 又是依靠超级块、inode、dentry 以及文件这些结构来发挥作用。所以文件系统的架构就体现在这些结构的使用方式中。

2.2.1 超级块作用分析

每个文件系统都有一个超级块结构，每个超级块都要链接到一个超级块链表。而文件系统内的每个文件在打开时都需要在内存分配一个 inode 结构，这些 inode 结构都要链接到超级块。

图 2-1 展示了超级块之间的关系以及超级块和 inode 之间的链接关系。super_block1 和 super_block2 是两个文件系统的超级块，它们被链接到 super_blocks 链表头，后者使用的就是内核基础层介绍的双向链表数据结构，顺着 super_blocks 链表可以遍历整个操作系统打开

过的文件的 inode 结构。

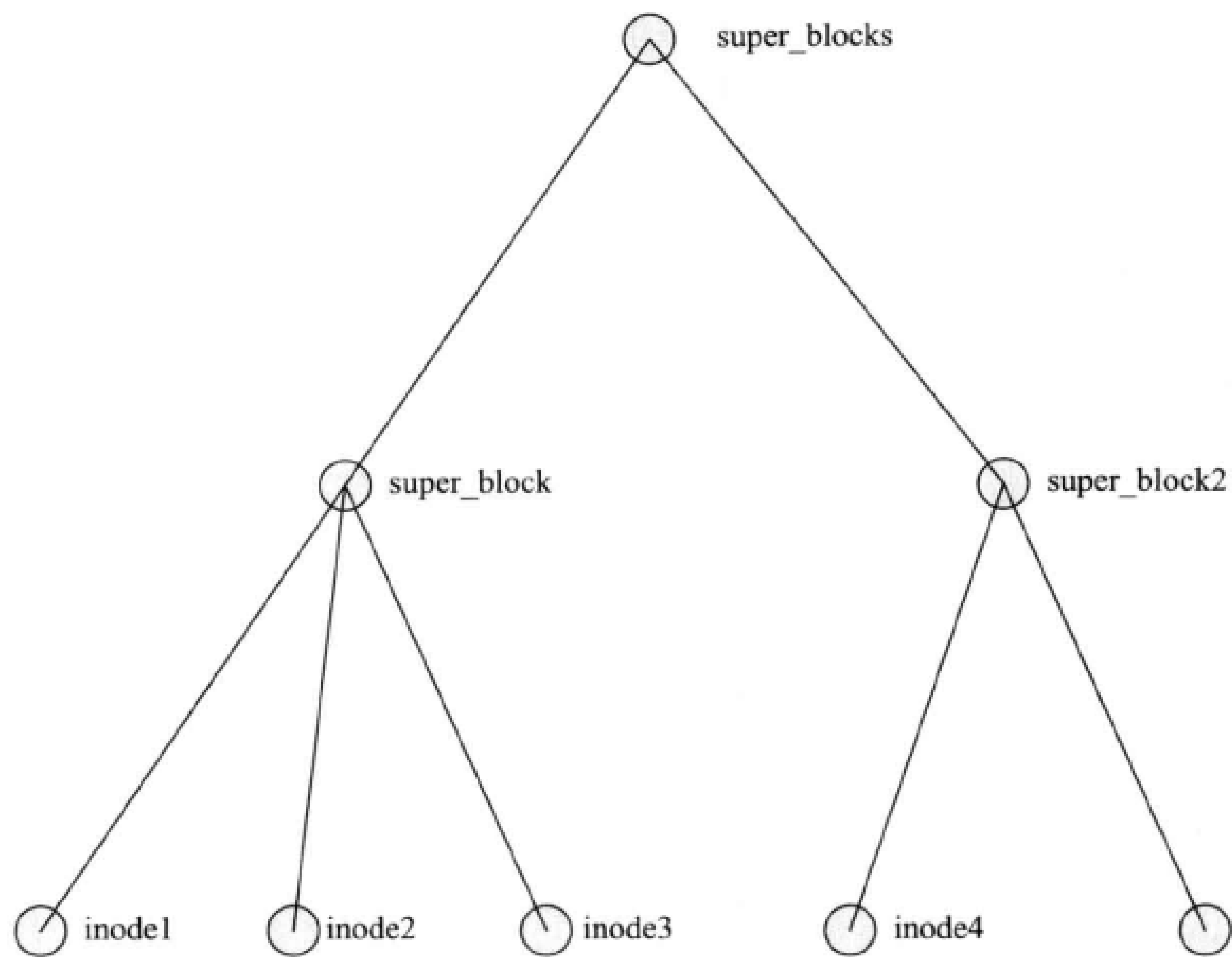


图 2-1 超级块的结构

2.2.2 dentry 作用分析

2.1.3 节分析了 VFS 中 dentry 的数据结构和作用，为了进一步理解 dentry，我们用图 2-2 来解释 dentry 的链接关系。

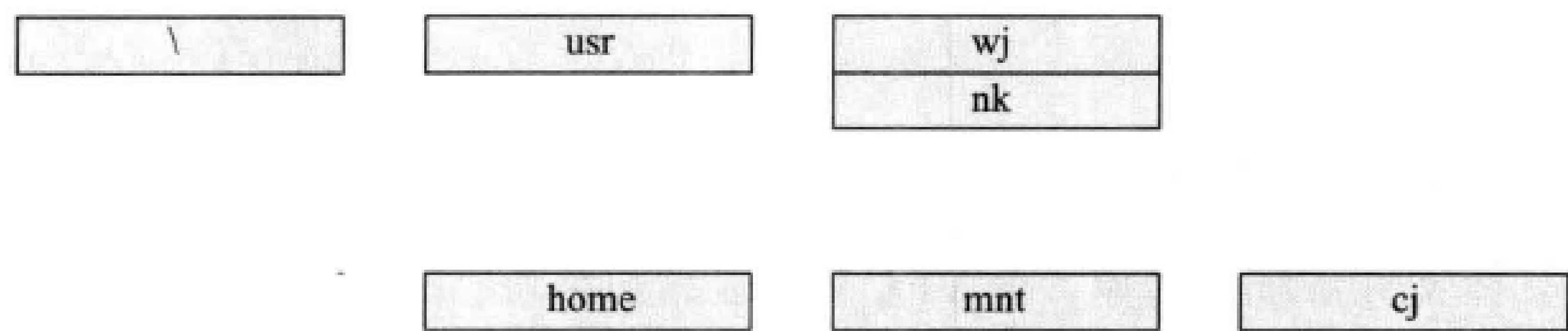


图 2-2 文件系统的目录结构

如图 2-2 所示，根目录下有 usr 和 home 两个目录，usr 目录下有 wj 和 nk 两个文件，home 目录下有个 mnt 目录，这是另外一个文件系统，挂载（mount）到当前文件系统。在 mnt 目录下有个 cj 文件。

文件系统的 dentry 链表图如图 2-3 所示。这只是个示意图，但是展示了几个重要的概念。

1）每个文件的 dentry 链接到父目录的 dentry，形成了文件系统的结构树。

具体说，就是 usr 和 home 两个 dentry 的 d_child 成员链接到根目录 dentry 的 d_subdirs 成员。而 wj 和 nk 两个 dentry 结构链接到 usr 这个 dentry。

2）所有的 dentry 都指向一个 dentry_hashtable。

dentry_hashtable 是个数组，它的数组成员是第 1 章介绍过的 hash 链表数据结构。这里所说的 dentry，指的是在内存中的 dentry。如果某个文件已经被打开过，内存中就应该有该文件的 dentry 结构，并且该 dentry 被链接到 dentry_hashtable 数组的某个 hash 链表头。后续再访问该文件的时候，就可以直接从 hash 链表里面找到，避免了再次读硬盘。这是 dentry 的 cache 概念。

3）home 目录下的 mnt 目录指向一个挂载的文件系统。

如何判断目录不是一个普通的目录，而是一个文件系统？这是 dentry 的 d_mounted 成员的功能。如果该成员不为 0，代表该 dentry 是个挂载点，有文件系统挂载，需要特殊处理。

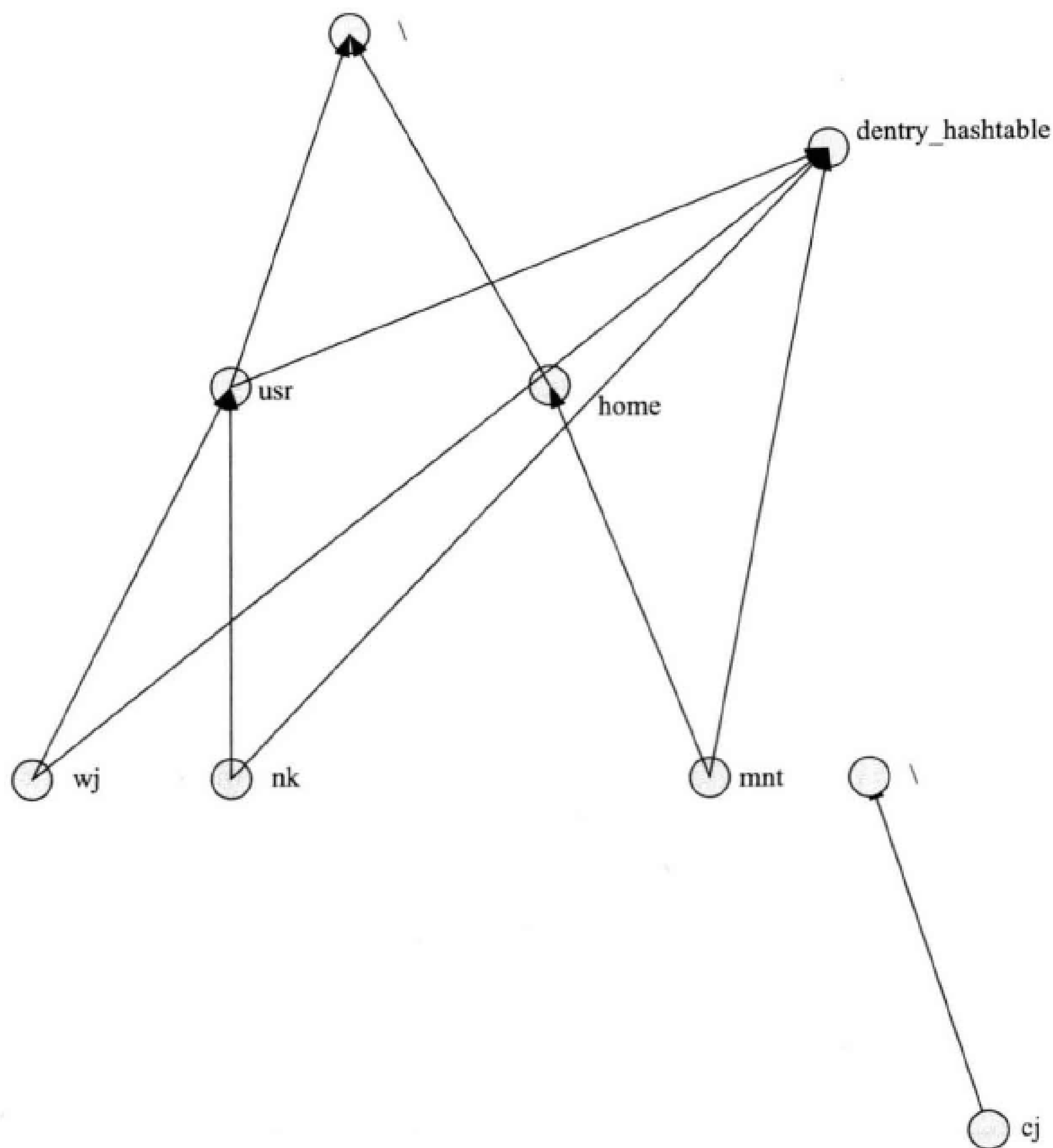


图 2-3 dentry 链表图

从图 2-3 可以看到，挂载过来的文件系统本身也有一个 dentry 树，也有自己的根目录。两个 dentry 树之间并没有链接关系。如何查找到挂载的文件系统哪？我们用图 2-4 来解释。

图 2-4 展示了一个新的数据结构 vfsmount。对这个数据结构不做过多的探讨，只需要知道每个文件系统都有这样一个结构。当文件系统被挂载的时候，它的 vfsmount 结构就被链接

到内核的一个全局链表——mount_hashtable数组链表。

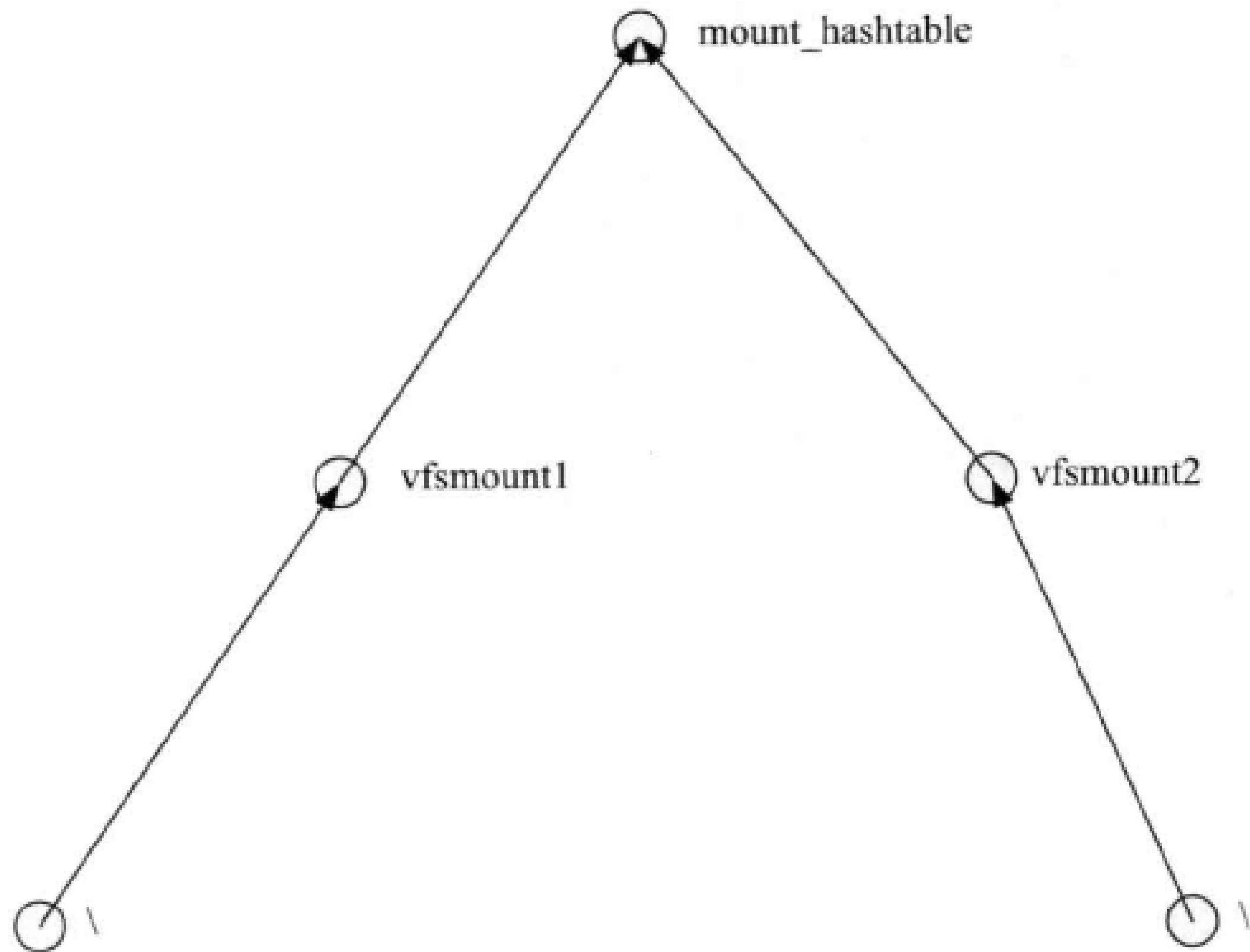


图 2-4 挂载的链表图

mount_hashtable是个数组，它的每个成员都是一个hash链表。上文的例子有两个vfsmount，cj文件所在文件系统的vfsmount被链接到mount_hashtable。这样当发现mnt目录是个特殊的目录时，从mount_hashtable数组找到hash链表头，再遍历整个hash链表，就能找到cj文件所在文件系统的vfsmount，然后mnt目录的dentry就被替换，置换为新文件系统的根目录。具体过程参考打开文件的代码分析。

2.2.3 inode作用分析

系统内核提供了一个hash链表数组inode_hashtable，所有的inode结构都要链接到数组里面的某个hash链表。这种用法和前文介绍的hash链表数组dentry_hashtable的用法很类似，这里就不再分析了。

在Linux系统里，字符设备和块设备、普通文件和目录、socket等都是一个文件，所以它们都有自己的inode结构。为了辨别这些不同的类型，inode结构的i_mode成员用不同的值代表不同的类型。如表2-1所示。

表 2-1 i_mode代表的类型

i-mode	类型
S_IFBLK	块设备
S_IFCHR	字符设备
S_IFDIR	目录
S_IFSOCK	socket
S_IFIFO	FIFO

inode 还有一个重要的作用是缓存文件的数据内容。这是通过成员 i_mapping 实现的。i_mapping 使用了数据结构 radix 树来保存文件的数据。这个数据结构在 1.2 节已经介绍过。在后面文件的读写过程分析中，还将继续分析文件内容读写的代码。

2.2.4 文件作用分析

文件对象代表进程和具体文件交互的关系。内核为每个打开的文件申请一个文件对象，同时返回文件的号码。如图 2-5 所示，每个进程都指向一个文件描述符表。

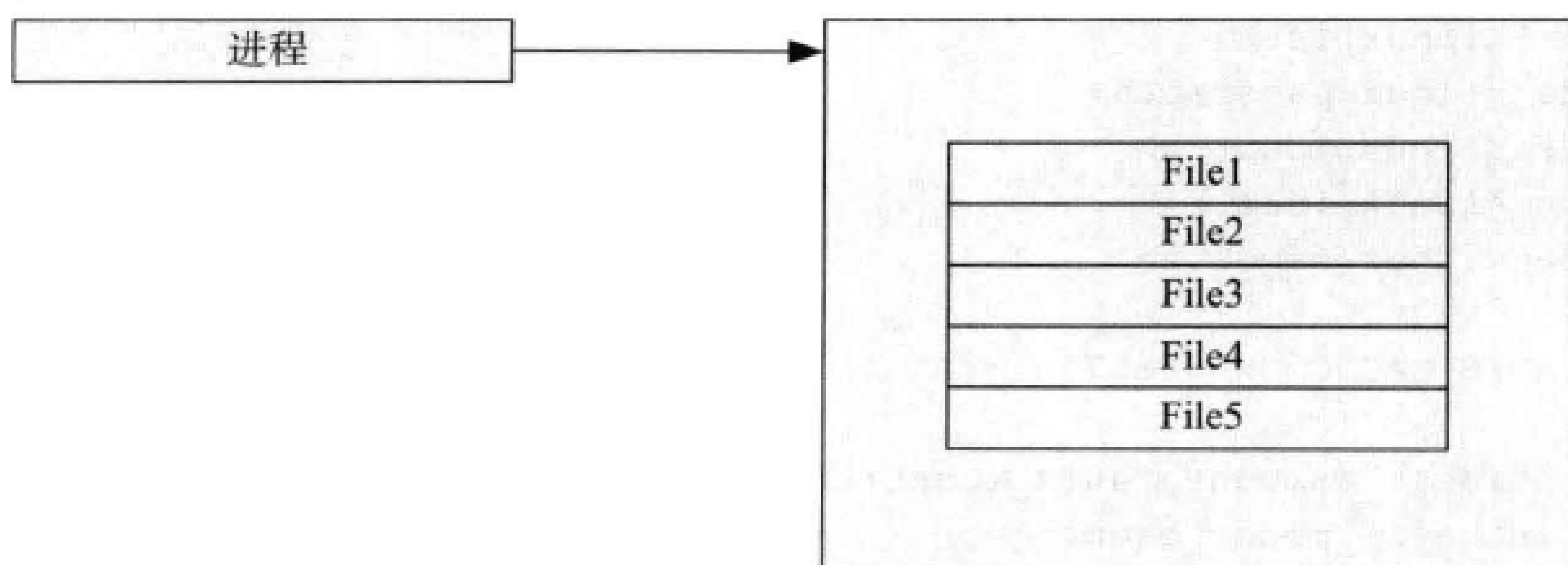


图 2-5 进程和文件描述符表

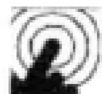
这个表里面用数组保存了进程中每个打开的文件。当应用进程打开一个硬盘上的文件时，内核要为这个文件分配一个文件对象并保存文件指针到文件描述符表里面的数组。

 **注意**

这里的“文件”指的是在硬盘上具体存在的文件，而“文件对象”是在内存里存在的文件结构。当读写文件的时候，通过文件号就可以获得文件的对象。这也就是读写文件必须先打开文件的原因，如果不执行打开的过程，是不能完成文件读写的。

2.3 从代码层次深入分析文件系统

文件系统千头万绪，但对用户来说，最重要的就是创建目录、创建文件、打开文件和文件读写。对于通常的硬盘文件系统来说，这要涉及硬盘的读写和硬盘空间管理，而读写从文件系统层一直到通用块设备层再到硬盘驱动，未免太过复杂。为了简化，我们给出一个最简单文件系统，通过这个例子导入文件系统的基本概念。然后通过代码分析，逐步深入内核，了解一下博大精深的内核架构。

 **提示**

本文假定读者已经了解模块概念，以及如何编译和安装模块。

2.3.1 一个最简单的文件系统 aufs

我们先写一个最简单的文件系统，这个文件系统直接创建在内存中。它在内存中创建了两个目录和几个文件，用户可以通过 ls 命令显示目录和文件，但是无法创建目录和文件，也不能对文件进行读写操作。这样不涉及硬盘操作，大大简化了开始阶段需要考虑的问题，这个例子如代码清单 2-5 所示。

代码清单 2-5 最简单文件系统 aufs 源代码

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/pagemap.h>
#include <linux/mount.h>
#include <linux/init.h>
#include <linux/namei.h>

#define AUFS_MAGIC 0x64668735

static struct vfsmount *aufs_mount;
static int aufs_mount_count;

static struct inode *aufs_get_inode(struct super_block *sb, int mode, dev_t dev)
{
  struct inode *inode = new_inode(sb);

  if (inode) {
        inode->i_mode = mode;
        inode->i_uid = current->fsuid;
        inode->i_gid = current->fsgid;
        inode->i_blksize = PAGE_CACHE_SIZE;
        inode->i_blocks = 0;
        inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
        switch (mode & S_IFMT) {
        default:
            init_special_inode(inode, mode, dev);
            break;
        case S_IFREG:
            printk("creat a  file \n");
            break;
        case S_IFDIR:
            inode->i_op = &simple_dir_inode_operations;
            inode->i_fop = &simple_dir_operations;
            printk("creat a dir file \n");

            inode->i_nlink++;
            break;
        }
  }
  return inode;
}
```

```
/* SMP-safe */
static int aufs_mknod(struct inode *dir, struct dentry *dentry,
           int mode, dev_t dev)
{
  struct inode *inode;
  int error = -EPERM;

  if (dentry->d_inode)
       return -EEXIST;

  inode = aufs_get_inode(dir->i_sb, mode, dev);
  if (inode) {
       d_instantiate(dentry, inode);
       dget(dentry);
       error = 0;
  }
  return error;
}

static int aufs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
  int res;

  res = aufs_mknod(dir, dentry, mode |S_IFDIR, 0);
  if (!res)
       dir->i_nlink++;
  return res;
}

static int aufs_create(struct inode *dir, struct dentry *dentry, int mode)
{
  return aufs_mknod(dir, dentry, mode | S_IFREG, 0);
}

static int aufs_fill_super(struct super_block *sb, void *data, int silent)
{
  static struct tree_descr debug_files[] = {{""}};

  return simple_fill_super(sb, AUFS_MAGIC, debug_files);
}

static struct super_block *aufs_get_sb(struct file_system_type *fs_type,
                      int flags, const char *dev_name,
                      void *data)
{
  return get_sb_single(fs_type, flags, data, aufs_fill_super);
}

static struct file_system_type au_fs_type = {
  .owner =     THIS_MODULE,
```

```
    .name =     "aufs",
    .get_sb =   aufs_get_sb,
    .kill_sb =  kill_litter_super,
};

static int aufs_create_by_name(const char *name, mode_t mode,
                  struct dentry *parent,
                  struct dentry **dentry)
{
  int error = 0;

  /* If the parent is not specified, we create it in the root.
   * We need the root dentry to do this, which is in the super
   * block. A pointer to that is in the struct vfsmount that we
   * have around.
   */
  if (!parent ) {
       if (aufs_mount && aufs_mount->mnt_sb) {
           parent = aufs_mount->mnt_sb->s_root;
       }
  }
  if (!parent) {
       printk("Ah! can not find a parent!\n");
       return -EFAULT;
  }

  *dentry = NULL;
  mutex_lock(&parent->d_inode->i_mutex);
  *dentry = lookup_one_len(name, parent, strlen(name));
  if (!IS_ERR(dentry)) {
       if ((mode & S_IFMT) == S_IFDIR)
            error = aufs_mkdir(parent->d_inode, *dentry, mode);
       else
            error = aufs_create(parent->d_inode, *dentry, mode);
  } else
       error = PTR_ERR(dentry);
  mutex_unlock(&parent->d_inode->i_mutex);

  return error;
}

struct dentry *aufs_create_file(const char *name, mode_t mode,
                   struct dentry *parent, void *data,
                   struct file_operations *fops)
{
  struct dentry *dentry = NULL;
  int error;

  printk("aufs: creating file '%s'\n",name);
```

```
  error = aufs_create_by_name(name, mode, parent, &dentry);
  if (error) {
       dentry = NULL;
       goto exit;
  }
  if (dentry->d_inode) {
       if (data)
            dentry->d_inode->u.generic_ip = data;
       if (fops)
            dentry->d_inode->i_fop = fops;
  }
exit:
  return dentry;
}

struct dentry *aufs_create_dir(const char *name, struct dentry *parent)
{
  return aufs_create_file(name,
                S_IFDIR | S_IRWXU | S_IRUGO | S_IXUGO,
                parent, NULL, NULL);
}

static int __init aufs_init(void)
{
  int retval;
       struct dentry *pslot;

  retval = register_filesystem(&au_fs_type);

  if (!retval) {
       aufs_mount = kern_mount(&au_fs_type);
       if (IS_ERR(aufs_mount)) {
           printk(KERN_ERR "aufs: could not mount!\n");
           unregister_filesystem(&au_fs_type);
           return retval;
       }
  }

  pslot = aufs_create_dir("woman star",NULL);
  aufs_create_file("lbb", S_IFREG | S_IRUGO, pslot, NULL, NULL);
  aufs_create_file("fbb", S_IFREG | S_IRUGO, pslot, NULL, NULL);
  aufs_create_file("ljl", S_IFREG | S_IRUGO, pslot, NULL, NULL);

  pslot = aufs_create_dir("man star",NULL);
  aufs_create_file("ldh", S_IFREG | S_IRUGO, pslot, NULL, NULL);
  aufs_create_file("lcw", S_IFREG | S_IRUGO, pslot, NULL, NULL);
  aufs_create_file("jw", S_IFREG | S_IRUGO, pslot, NULL, NULL);

  return retval;
}
```

```
static void __exit aufs_exit(void)
{
  simple_release_fs(&aufs_mount, &aufs_mount_count);
  unregister_filesystem(&au_fs_type);
}

module_init(aufs_init);
module_exit(aufs_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("This is a simple module");
MODULE_VERSION("Ver 0.1");
```

整个程序只有两百多行，即使对文件系统一点不懂也能了解大概意思。这个例子不包括文件读写等内容，目的只是说明文件系统内 dentry、inode、super_block 等几个重要概念。

程序编译后，我们通过 insmod 命令加载模块，然后执行如下操作。

1) 在根目录下创建一个目录：

```
mkdir au
```

2）挂载文件系统：

```
mount -t aufs none /au
```

3）列出文件系统的内容：

```
ls
```

看到了什么？可以发现“woman star”和“man star”两个目录。然后到 woman star 目录再执行 ls，目录下果然有 lbb、fbb、ljl 三个文件。这就是我们在代码中希望做到的事情。

2.3.2 文件系统如何管理目录和文件

现在我们先看看这段程序究竟执行了什么就创建了一个最简单文件系统，然后再深入内核分析究竟发生了什么。

整个程序代码可以分为三部分。1）首先是 register_filesystem 函数，这个函数把 aufs 文件系统登记到系统；2）然后调用 kern_mount 函数为文件系统申请必备的数据结构；3）最后在 aufs 文件系统内创建两目录，每个目录下面创建三个文件。

register_filesystem 函数顾名思义，就是把一个特定的文件系统登记到内核。这个函数的实现如代码清单 2-6 所示。

代码清单 2-6 register_filesystem 函数

```
int register_filesystem(struct file_system_type * fs)
{
  int res = 0;
  struct file_system_type ** p;

  if (!fs)
       return -EINVAL;
```

```
    if (fs->next)
        return -EBUSY;
    INIT_LIST_HEAD(&fs->fs_supers);
    write_lock(&file_systems_lock);
    p = find_filesystem(fs->name);
    if (*p)
        res = -EBUSY;
    else
        *p = fs;
    write_unlock(&file_systems_lock);
    return res;
}
```

函数 register_filesystem 的参数是一个文件类型指针，函数执行部分要在内核寻找相同名字的文件系统，如果不存在相同名字的文件系统，就把 aufs 加入到系统的文件系统链表。如果这个文件系统已经存在，则返回忙。

内核定义一个全局变量 file_systems，用来保存所有登记的文件系统，而 find_filesystem 利用全局变量 file_systems 执行了具体的查找过程。这个函数非常简单，但是显然主要工作没在这里完成，我们继续往下看 kern_mount 函数。kern_mount 真正为文件系统分配了超级块对象和 vfsmount 对象。具体代码如代码清单 2-7 所示。

代码清单 2-7　kern_mount 函数

```
struct vfsmount *kern_mount(struct file_system_type *type)
{
    return vfs_kern_mount(type, 0, type->name, NULL);
}
```

kern_mount 只是 vfs_kern_mount 的封装。vfs_kern_mount 的代码如代码清单 2-8 所示。

代码清单 2-8　vfs_kern_mount 函数

```
struct vfsmount *
vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
{
    struct vfsmount *mnt;
    char *secdata = NULL;
    int error;

    if (!type)
        return ERR_PTR(-ENODEV);
      /* 分配一个 vfsmount 结构 */
    error = -ENOMEM;
    mnt = alloc_vfsmnt(name);
    if (!mnt)
        goto out;
```

vfs_kern_mount 函数的代码很长，为了方便阅读，我们将 vfs_kern_mount 分为三个部分。

第一部分是根据文件系统的名字为文件系统创建了一个 vfsmount 结构。这个结构在 2.2.2 节分析过，是用来为文件系统之间的挂载关系而设计的，后文还将继续分析具体的 mount 过程。

```
/* 如果 mount 有数据参数，才执行下面的代码。对本章的例子来说，这部分跳过 */
  if (data) {
        secdata = alloc_secdata();
        if (!secdata)
            goto out_mnt;

        error = security_sb_copy_data(type, data, secdata);
        if (error)
            goto out_free_secdata;
  }

    /* 调用文件系统超级块提供的 get_sb 函数。对 aufs 文件系统来说，就是 aufs_get_sb*/
  error = type->get_sb(type, flags, name, data, mnt);
  if (error < 0)
        goto out_free_secdata;
```

vfs_kern_mount 的第二部分是调用文件系统提供的 get_sb 创建一个超级块对象。创建超级块对象的同时，还要创建一个 dentry 结构作为文件系统的根 dentry（root dentry）和一个 inode 结构作为文件系统的根 inode。文件系统的根 dentry 等于文件系统的根目录，后续创建的一级目录都要以这个根目录作为父目录。

```
    /* 安全相关的代码，可以跳过 */
  error = security_sb_kern_mount(mnt->mnt_sb, secdata);
  if (error)
        goto out_sb;

    /*mnt 数据在申请的时候被赋予空值，这里 mnt_mountpoint 和 mnt_root 都为空 */
  mnt->mnt_mountpoint = mnt->mnt_root;
  mnt->mnt_parent = mnt;
  up_write(&mnt->mnt_sb->s_umount);
  free_secdata(secdata);
  return mnt;
}
```

最后一部分设置 vfsmount 结构的父指针为自身，mnt_mountpoint 为文件系统的根 dentry。如果把文件系统 mount 到其他的文件系统，那么这两个参数就要设置为源文件系统的参数。

三部分综合起来，vfs_kern_mount 函数实际已经执行了文件系统登记的大部分工作。

vfs_kern_mount 函数的整体分析完毕，其中的 get_sb 函数比较重要，需要细节分析。对 aufs 文件系统而言，它的 get_sb 函数是 aufs_get_sb。这个函数的代码如代码清单 2-9 所示。

代码清单 2-9 aufs_get_sb 函数

```
static struct super_block *aufs_get_sb(struct file_system_type *fs_type,
                    int flags, const char *dev_name,void *data)
```

```
{
  return get_sb_single(fs_type, flags, data, aufs_fill_super);
}
```

aufs_get_sb 是个封装函数，实际上调用了系统提供的 get_sb_single 函数，这个函数的代码如代码清单 2-10 所示。

代码清单 2-10 get_sb_single 函数

```
        int get_sb_single(struct file_system_type *fs_type,
791     int flags, void *data,
792     int (*fill_super)(struct super_block *, void *, int),
793     struct vfsmount *mnt)
794     {
795     struct super_block *s;
796     int error;
797
798     s = sget(fs_type, compare_single, set_anon_super, NULL);
799     if (IS_ERR(s))
800          return PTR_ERR(s);
801     if (!s->s_root) {
802          s->s_flags = flags;
             /* 调用传递进来的函数指针。这里就是 aufs_fill_super*/
803          error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);
             ……
809          s->s_flags |= MS_ACTIVE;
810     }
        /* 改变 mount 的选项 */
811     do_remount_sb(s, flags, data, 0);
812     return simple_set_mnt(mnt, s);
813 }
```

get_sb_single 首先获得一个超级块对象，如果文件系统的超级块对象已经存在，返回对象指针，如果不存在，则创建一个新的超级块对象。

创建超级块对象后，第 801 行代码检查超级块对象是否有根 dentry，如果尚未有根 dentry，调用传递进来的函数指针 fill_super 为超级块对象填充根 dentry 和根 inode。

最后的 simple_set_mnt 函数要把创建的超级块对象赋值给 vfsmount 结构所指的超级块，同时 vfsmount 所指的 mnt_root 点赋值为超级块所指的根 dentry。于是从 vfsmount 结构就可以获得文件系统的超级块对象和根 dentry。

函数 fill_super 函数作用是为超级块对象申请必备的成员。对于 aufs 文件系统而言，传递的函数指针是 aufs_fill_super 函数。这个函数的代码如代码清单 2-11 所示。

代码清单 2-11 aufs_fill_super 函数

```
static int aufs_fill_super(struct super_block *sb, void *data, int silent)
{
  static struct tree_descr debug_files[] = {{""}};
```

```
    return simple_fill_super(sb, AUFS_MAGIC, debug_files);
}
```

aufs_fill_super 定义了一个空的 tree_descr 结构，这个结构的作用是描述一些文件。如果不为空，填充超级块的同时，需要在根目录下创建一些文件；当前为空，说明不需要创建任何文件。simple_fill_super 函数实现了填充根 dentry 的执行过程，具体代码如代码清单 2-12 所示。

代码清单 2-12 填充超级块的 simple_fill_super 函数

```
     int simple_fill_super(struct super_block *s, int magic, struct tree_descr *files)
368  {
369  static struct super_operations s_ops = {.statfs = simple_statfs};
370  struct inode *inode;
371  struct dentry *root;
372  struct dentry *dentry;
373  int i;
374
375  s->s_blocksize = PAGE_CACHE_SIZE;
376  s->s_blocksize_bits = PAGE_CACHE_SHIFT;
377  s->s_magic = magic;
378  s->s_op = &s_ops;
379  s->s_time_gran = 1;
380
381  inode = new_inode(s);
382  if (!inode)
383      return -ENOMEM;
384  inode->i_mode = S_IFDIR | 0755;
385  inode->i_uid = inode->i_gid = 0;
386  inode->i_blksize = PAGE_CACHE_SIZE;
387  inode->i_blocks = 0;
388  inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
389  inode->i_op = &simple_dir_inode_operations;
390  inode->i_fop = &simple_dir_operations;
391  inode->i_nlink = 2;
     /* 创建一个 dentry 对象 */
392  root = d_alloc_root(inode);
```

simple_fill_super 函数第一部分是为超级块对象赋初值。超级块对象指示了文件系统的块大小，第 375 行赋值 aufs 文件系统的块尺寸为一个页面，一个页面通常是 4K 大小，以比特位计算就是 12 位。第 378 行为超级块对象赋予操作函数，simple_fill_super 实际上只为超级块对象提供了一个操作函数 simple_statfs。

从第 381 行开始创建一个 inode 结构。这个 inode 是文件系统的根 inode，所以第 382 行设置它是一个目录，代表根目录，然后设置它的块尺寸和文件大小。inode 结构包含它的操作函数，第 389 行和 390 行分别赋值它的 inode 操作函数和文件操作函数。

第 392 行 d_alloc_root 作用是创建一个根 dentry。根 dentry 的父结构是自身，它指向的

超级块对象就是文件系统的超级块对象。根 dentry 的名字被指定为斜杠符“/”，这就是从根目录开始查找文件使用斜杠符“/”作为开始的原因。

```
    for (i = 0; !files->name || files->name[0]; i++, files++) {
398 if (!files->name)
399    continue;
400    dentry = d_alloc_name(root, files->name);
401 if (!dentry)
402     goto out;
403 inode = new_inode(s);
404 if (!inode)
405     goto out;
406 inode->i_mode = S_IFREG | files->mode;
407 inode->i_uid = inode->i_gid = 0;
408 inode->i_blksize = PAGE_CACHE_SIZE;
409 inode->i_blocks = 0;
410 inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
411 inode->i_fop = files->ops;
412 inode->i_ino = i;
413 d_add(dentry, inode);
414 }
    /* 超级块内保存 root dentry 的指针。这样内核就可以通过 root dentry 遍历文件系统了 */
415 s->s_root = root;
416 return 0;
```

simple_fill_super 函数的第二部分是根据传递进来的参数，在根目录下创建一系列的文件。因为当前场景传进来的是空参数，所以这里可以跳过。

通过代码分析，到目前为止，创建了一个超级块对象，创建了一个根 dentry 和一个根 inode。后面创建的文件和目录都应该链接到这个根 dentry。代码里面的 new_inode 和 d_alloc_root 都很简单，留给读者自己分析。

再回到 aufs 文件系统。创建目录和创建文件调用了不同的函数，aufs_create_dir 函数的作用是创建一个目录，它的代码如代码清单 2-13 所示。

代码清单 2-13 aufs_create_dir

```
struct dentry *aufs_create_dir(const char *name, struct dentry *parent)
{
  return aufs_create_file(name,
                S_IFDIR | S_IRWXU | S_IRUGO | S_IXUGO,
                parent, NULL, NULL);
}
```

aufs_create_dir 是 aufs_create_file 函数的封装函数，aufs_create_file 函数执行创建文件的过程。调用 aufs_create_file 的时候，传递的参数 S_IFDIR 指明创建的是一个目录。

aufs_create_file 的代码如代码清单 2-14 所示。

代码清单 2-14 创建文件的 aufs_create_file 函数

```
struct dentry *aufs_create_file(const char *name, mode_t mode,
                  struct dentry *parent, void *data,
                  struct file_operations *fops)
{
  ......
  error = aufs_create_by_name(name, mode, parent, &dentry);
  ......
  if (dentry->d_inode) {
        if (data)
            dentry->d_inode->u.generic_ip = data;
        if (fops)
            dentry->d_inode->i_fop = fops;
  }
exit:
  return dentry;
}
```

回忆前面的知识，文件是由 dentry 和 inode 代表的。而执行函数 aufs_create_by_name 后，空指针 dentry 已经被赋值了，而且有 d_inode 成员，说明它的的作用是创建文件的 dentry 和 inode 结构，如代码清单 2-15 所示。

代码清单 2-15 根据名字创建文件的 aufs_create_by_name 函数

```
static int aufs_create_by_name(const char *name, mode_t mode,
                 struct dentry *parent,
                 struct dentry **dentry)
{
  int error = 0;

  /* 如果没有父目录，就赋予一个。赋予的是哪一个？就是前面创建的 root dentry*/
  if (!parent ) {
      if (aufs_mount && aufs_mount->mnt_sb) {
          parent = aufs_mount->mnt_sb->s_root;
      }
  }
  if (!parent) {
      printk("Ah! can not find a parent!\n");
      return -EFAULT;
}
```

函数 aufs_create_by_name 起始部分要判断是否有父目录，如果没有父目录就赋予一个。赋予的是哪一个？就是文件系统的根 dentry。

```
*dentry = NULL;
  mutex_lock(&parent->d_inode->i_mutex);
  /* 检查要创建的这个目录存在不存在？是不是重复了 */
  *dentry = lookup_one_len(name, parent, strlen(name));
  if (!IS_ERR(dentry)) {
```

```
        /* 分成两个分支。如果是目录则调用 aufs_mkdir，如果是文件，则调用 aufs_create*/
        if ((mode & S_IFMT) == S_IFDIR)
             error = aufs_mkdir(parent->d_inode, *dentry, mode);
        else
             error = aufs_create(parent->d_inode, *dentry, mode);
    } else
        error = PTR_ERR(dentry);
    mutex_unlock(&parent->d_inode->i_mutex);

    return error;
}
```

然后调用 lookup_one_len 获得一个 dentry 结构。lookup_one_len 函数首先在父目录下根据名字查找 dentry 结构，如果存在同名的 dentry 结构就返回指针，如果不存在就创建一个 dentry。lookup_one_len 的代码如代码清单 2-16 所示。

代码清单 2-16 查找同名 dentry 的 lookup_one_len 函数

```
     struct dentry * lookup_one_len(const char * name, struct dentry * base, int len)
1272 {
     ....../* 省略参数定义 */
1277 this.name = name;
1278 this.len = len;
1279 if (!len)
1280     goto access;
1281  /* 这里根据名字计算 hasn 值，看看是怎么计算的 */
1282 hash = init_name_hash();
1283 while (len--) {
1284     c = *(const unsigned char *)name++;
1285     if (c == '/' || c == '\0')
1286         goto access;
1287     hash = partial_name_hash(c, hash);
1288 }
1289 this.hash = end_name_hash(hash);
1290
1291 return __lookup_hash(&this, base, NULL);
```

第 1282 ~ 1289 行的作用是计算名字的 hash 值。因为初始 hash 函数 init_name_hash 和最终 hash 函数 end_name_hash 都不执行任何的计算，所以最终得到的 hash 值是将名字中的每个字符做固定的数学运算得到的。

__lookup_hash 函数是通过 hash 值查找同名字的 dentry 结构，它的代码如代码清单 2-17 所示。

代码清单 2-17 在 hash 链表中查找的 __lookup_hash 函数

```
static struct dentry * __lookup_hash(struct qstr *name,
struct dentry * base, struct nameidata *nd)
{
  struct dentry * dentry;
```

```
    struct inode *inode;
    int err;

    inode = base->d_inode;
    err = permission(inode, MAY_EXEC, nd);
    dentry = ERR_PTR(err);
    if (err)
        goto out;

    /*
     * See if the low-level filesystem might want
     * to use its own hash..
     */
    if (base->d_op && base->d_op->d_hash) {
        err = base->d_op->d_hash(base, name);
        dentry = ERR_PTR(err);
        if (err < 0)
            goto out;
    }
```

__lookup_hash 函数第一部分是检查 inode 的权限，然后检查文件系统是否提供了特有的 hash 函数，如果有 hash 函数，则调用重新计算 hash 值。

```
dentry = cached_lookup(base, name, nd);
if (!dentry) {
        /* 如果没有找到。说明这个目录不存在，则创建一个 dentry*/
        struct dentry *new = d_alloc(base, name);
        dentry = ERR_PTR(-ENOMEM);
        if (!new)
            goto out;
        dentry = inode->i_op->lookup(inode, new, nd);
        if (!dentry)
            dentry = new;
        else
            dput(new);
    }
out:
    return dentry;
}
```

cached_lookup 在 dentry cache 里面查找同名的 dentry 结构，如果返回为空，说明不存在同名的 dentry 结构，那么调用 d_alloc 创建一个新的 dentry 结构。

创建 dentry 结构完成后，需要再次调用文件系统的 lookup 查找是否有同名的 dentry 存在。这样做为了防止同名的 dentry 已经被其他用户提前创建了。

cached_lookup 函数的代码如代码清单 2-18 所示。

代码清单 2-18 cached_lookup 函数

```
static struct dentry * cached_lookup(struct dentry * parent,
```

```
struct qstr * name, struct nameidata *nd)
{
  struct dentry * dentry = __d_lookup(parent, name);

  /* lockess __d_lookup may fail due to concurrent d_move()
   * in some unrelated directory, so try with d_lookup
   */
  /* 注意原来的解释，为何再次查找？因为要防止一个并发的 move 操作 */
  if (!dentry)
       dentry = d_lookup(parent, name);
       ....../* 省略校验的代码 */
}
```

cached_lookup 函数使用两次 lookup。第一次调用 __d_lookup，而第二次调用 d_lookup。这是因为要防止并发的 d_move 操作。__d_lookup 执行中没有获得重命名锁，有可能因为重命名操作而失败。

这种类似的做法在内核里很常见。因为内核是为所有用户进程提供服务，必须考虑并发性。

__d_lookup 的代码如代码清单 2-19 所示。

代码清单 2-19 __d_lookup 函数

```
struct dentry * __d_lookup(struct dentry * parent, struct qstr * name)
{
  unsigned int len = name->len;
  unsigned int hash = name->hash;
  const unsigned char *str = name->name;
  struct hlist_head *head = d_hash(parent,hash);
  struct dentry *found = NULL;
  struct hlist_node *node;
  struct dentry *dentry;
```

函数 __d_lookup 起始部分要找到 hash 链表。内核中的 dentry 结构都根据 hash 值链接到众多 hash 链表中，这些 hash 链表的头结构保存在数组 dentry_hashtable 中。利用 parent 指针和 hash 值计算最终的 hash 值，从数组 dentry_hashtable 中获得 hash 链表的链表头。

```
  rcu_read_lock();

  /* 遍历 hash list */
  hlist_for_each_entry_rcu(dentry, node, head, d_hash) {
      struct qstr *qstr;

      if (dentry->d_name.hash != hash)
          continue;
      if (dentry->d_parent != parent)
          continue;

      spin_lock(&dentry->d_lock);
```

```
        /*
         * Recheck the dentry after taking the lock - d_move may have
         * changed things.  Don't bother checking the hash because we're
         * about to compare the whole name anyway.
         */
        if (dentry->d_parent != parent)
            goto next;

        /*
         * It is safe to compare names since d_move() cannot
         * change the qstr (protected by d_lock).
         */
        qstr = &dentry->d_name;
        /* 如果文件系统定义了 d_compare 函数，则调用 */
        if (parent->d_op && parent->d_op->d_compare) {
            if (parent->d_op->d_compare(parent, qstr, name))
                goto next;
        } else {
            /* 比较长度是否相同 */
            if (qstr->len != len)
                goto next;
            /* 比较目录名字是否相同 */
            if (memcmp(qstr->name, str, len))
                goto next;
        }

        if (!d_unhashed(dentry)) {
            atomic_inc(&dentry->d_count);
            found = dentry;
        }
        spin_unlock(&dentry->d_lock);
        break;
    next:
        spin_unlock(&dentry->d_lock);
    }
    rcu_read_unlock();

    return found;
}
```

获得 hash 链表头之后，__d_lookup 遍历整个 hash 链表，寻找匹配的 dentry 结构。匹配的过程参考代码中的注释部分。找到或者创建 dentry 结构后，还需要创建 inode 结构。接续前面的分析，返回 aufs_create_by_name 函数。创建 inode 结构是其中 aufs_mkdir 函数的功能，它的代码如代码清单 2-20 所示。

代码清单 2-20 创建目录文件的 aufs_mkdir 函数

```
static int aufs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
  int res;
```

```
    /* 参数 S_IFDIR 指示是创建一个目录文件的 inode*/
    res = aufs_mknod(dir, dentry, mode |S_IFDIR, 0);
    if (!res)
        dir->i_nlink++;
    return res;
}
```

aufs_mkdir 封装了 aufs_mknod 函数，aufs_mknod 代码如代码清单 2-21 所示。

代码清单 2-21　aufs_mknod 函数

```
static int aufs_mknod(struct inode *dir, struct dentry *dentry,
            int mode, dev_t dev)
{
    struct inode *inode;
    int error = -EPERM;
    /* 如果 dentry 已经有 d_inode 结构，说明 inode 已经存在了 */
    if (dentry->d_inode)
        return -EEXIST;

    inode = aufs_get_inode(dir->i_sb, mode, dev);
    if (inode) {
        d_instantiate(dentry, inode);
        dget(dentry);
        error = 0;
    }
    return error;
}
```

aufs_mknod 函数通过 aufs_get_inode 来创建目录文件的 inode，然后调用 d_instantiate 函数把 dentry 加入到 inode 的 dentry 链表头。

aufs_get_inode 创建了一个 inode 结构，同时要把 inode 结构加入超级块对象的 inode 链表头，这样从超级块对象的 inode 链表，可以遍历文件系统内所有的 inode 结构。除了超级块对象的链表，inode 还要加入一个全局变量链表 inode_in_use，这个链表指示 inode 结构是活跃的，处于使用中。aufs_get_inode 的代码如代码清单 2-22 所示。

代码清单 2-22　aufs_get_inode 函数

```
static struct inode *aufs_get_inode(struct super_block *sb, int mode, dev_t dev)
{
    /* 申请一个 inode 结构 */
    struct inode *inode = new_inode(sb);

    if (inode) {
        inode->i_mode = mode;
        inode->i_uid = current->fsuid;
        inode->i_gid = current->fsgid;
        inode->i_blksize = PAGE_CACHE_SIZE;
        inode->i_blocks = 0;
```

```
        inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
        switch (mode & S_IFMT) {
        default:
            init_special_inode(inode, mode, dev);
            break;
        case S_IFREG:
            printk("creat a  file \n");
            break;
        case S_IFDIR:
            inode->i_op = &simple_dir_inode_operations;
            inode->i_fop = &simple_dir_operations;
            printk("creat a dir file \n");

            inode->i_nlink++;
            break;
        }
    }
    return inode;
}
```

可以看到，inode 结构的用户 ID 和组 ID 分别赋值为当前进程的文件系统用户 ID 和组 ID。代码中的 current 是内核定义的一个宏，它的作用是获得当前进程的结构指针。

aufs_get_inode 要根据 inode 的类型，设置不同的操作函数。对于特殊的 inode，比如块设备文件或者字符设备文件，调用 init_special_inode 赋值。对于目录文件，分别赋值 i_op 和 i_fop。

目前为止，整个 aufs 文件系统的代码，已经为每个文件和目录都创建了 dentry 结构，同时为每个文件和目录创建了 inode 结构。这些文件和目录的 dentry 结构层层链接，已经在内核形成了一颗 dentry 树。但是这颗树还不能被访问，要真正使用起来，还需要挂载到根文件系统。

2.3.3 文件系统的挂载过程

挂载（mount）做了什么事情？从理论上推断，系统本身也有一个文件目录树，如果把 aufs 创建的 dentry 树绑定到系统本来的 dentry 树上并建立链接，就可以从原先的系统树遍历到 aufs 的 dentry 树了。这就是 mount 过程。

执行文件系统的 mount 命令，要指定一个源文件系统和一个目的文件系统。同时要为目的文件系统指定一个目录，源文件系统就挂载到目的文件系统的这个目录下，这个目录称为**挂载点**。一般源文件系统要指定设备名，这个设备就是源文件系统所存在的设备。因为 aufs 文件系统只存在于内存，并不存在于硬盘设备，所以 aufs 不用指定设备名。

文件系统挂载通过系统调用 sys_mount 来执行，sys_mount 又调用 do_mount。do_mount 首先获得挂载点目录的 dentry 结构以及目的文件系统的 vfsmount 结构，这些信息保存在一个 nameidata 结构中。然后根据 mount 选项调用不同的函数执行 mount。因为 aufs 文件系统第一次执行 mount，所以调用的是 do_new_mount 函数，该函数执行了 mount 的大部分事情，它的代码如代码清单 2-23 所示。

代码清单 2-23 do_new_mount 函数

```
static int do_new_mount(struct nameidata *nd, char *type, int flags,
            int mnt_flags, char *name, void *data)
{
  struct vfsmount *mnt;

  if (!type || !memchr(type, 0, PAGE_SIZE))
      return -EINVAL;

  /* we need capabilities... */
  if (!capable(CAP_SYS_ADMIN))
      return -EPERM;
  mnt = do_kern_mount(type, flags, name, data);
  if (IS_ERR(mnt))
      return PTR_ERR(mnt);

  return do_add_mount(mnt, nd, mnt_flags, NULL);
}
```

do_kern_mount 前文已经分析了，目的是创建超级块对象和 root dentry 和 inode。因为 aufs 文件系统初始化时已经创建了这些对象，所以得到的是已经存在的对象。

do_add_mount 把源文件系统挂载到目的文件系统，它的代码如代码清单 2-24 所示。

代码清单 2-24 do_add_mount 函数

```
     int do_add_mount(struct vfsmount *newmnt, struct nameidata *nd,
1093         int mnt_flags, struct list_head *fslist)
1094 {
1095  int err;
1096
1097  down_write(&namespace_sem);
1098  /* Something was mounted here while we slept */
1099  while (d_mountpoint(nd->dentry) && follow_down(&nd->mnt, &nd->dentry))
1100      ;
1101  err = -EINVAL;
      /* 检查 namespace 是否当前进程的 namespace*/
1102  if (!check_mnt(nd->mnt))
1103      goto unlock;
1104
1105  /* Refuse the same filesystem on the same mount point */
1106  err = -EBUSY;
1107  if (nd->mnt->mnt_sb == newmnt->mnt_sb &&
1108     nd->mnt->mnt_root == nd->dentry)
1109      goto unlock;
1110
1111  err = -EINVAL;
1112  if (S_ISLNK(newmnt->mnt_root->d_inode->i_mode))
1113      goto unlock;
```

do_add_mount 函数第一部分是检查参数。

第1099行检查挂载点目录本身是否为挂载点。如果挂载点目录本身已经被挂载了（这通过检查挂载点目录的 dentry 结构的 d_mounted 成员是否不为0实现的），说明挂载点目录已经被挂载，那么要调用 follow_down 函数来找到真正的 dentry 结构和 vfsmount 对象。follow_down 的查找过程后文还会分析到，此处略过。

第1107行检查同一个文件系统是否已经在挂载点目录挂载了，如果已经挂载，要返回错误。

第1112行检查源文件系统根 inode（根 dentry 的 inode）是否符号链接，如果是符号链接，则返回错误。

```
1115 newmnt->mnt_flags = mnt_flags;
1116 if ((err = graft_tree(newmnt, nd)))
1117      goto unlock;
1118 /* 当前场景的 fslist 为 NULL，跳过 */
1119 if (fslist) {
1120      /* add to the specified expiration list */
1121      spin_lock(&vfsmount_lock);
1122      list_add_tail(&newmnt->mnt_expire, fslist);
1123      spin_unlock(&vfsmount_lock);
1124 }
1125 up_write(&namespace_sem);
1126 return 0;
```

graft_tree 函数把 aufs 的 dentry 树和目的文件系统的 dentry 树嫁接到一起，它的代码如代码清单 2-25 所示。

代码清单 2-25 graft_tree 函数

```
855  static int graft_tree(struct vfsmount *mnt, struct nameidata *nd)
856  {
857  int err;
858  if (mnt->mnt_sb->s_flags & MS_NOUSER)
859      return -EINVAL;
860
861  if (S_ISDIR(nd->dentry->d_inode->i_mode) !=
862       S_ISDIR(mnt->mnt_root->d_inode->i_mode))
863       return -ENOTDIR;
864
865  err = -ENOENT;
866  mutex_lock(&nd->dentry->d_inode->i_mutex);
     /* 检查挂载点目录是否是废弃的目录 */
867  if (IS_DEADDIR(nd->dentry->d_inode))
868      goto out_unlock;
869  ……/* 省略无关代码 */
874  err = -ENOENT;
875  if (IS_ROOT(nd->dentry) || !d_unhashed(nd->dentry))
876      err = attach_recursive_mnt(mnt, nd, NULL);
```

第 858 行判断源文件系统是否可以被挂载。Linux 系统的一些特殊的文件系统是不能被挂载的，比如 pipefs 文件系统、块设备文件系统（参见第 9 章）等。

第 861 行检查挂载点目录是否是一个目录文件，以及源文件系统的根 inode 是否目录文件。这两者都应该是目录才能执行挂载操作，如果不是目录则返回错误。

第 875 行检查挂载点是否有效。有效的条件是挂载点目录是一个根目录，或者挂载点目录被缓存在 dentry cache 中。这是因为文件系统根目录没有被缓存在 dentry cache 中，所以做这一步检查。检查通过，调用 attach_recursive_mnt 函数执行挂载操作。注意，此时 mnt 参数是 aufs 文件系统的 vfs_mount 对象，作为源文件系统；而 nd 参数保存了目的文件系统的 dentry 和 vfsmount 对象，作为目的点。attach_recursive_mnt 的代码如代码清单 2-26 所示。

代码清单 2-26 attach_recursive_mnt 函数

```
static int attach_recursive_mnt(struct vfsmount *source_mnt,
            struct nameidata *nd, struct nameidata *parent_nd)
{
  LIST_HEAD(tree_list);
  struct vfsmount *dest_mnt = nd->mnt;
  struct dentry *dest_dentry = nd->dentry;
  struct vfsmount *child, *p;

  if (propagate_mnt(dest_mnt, dest_dentry, source_mnt, &tree_list))
      return -EINVAL;
  /* 处理 shared*/
  if (IS_MNT_SHARED(dest_mnt)) {
      for (p = source_mnt; p; p = next_mnt(p, source_mnt))
          set_mnt_shared(p);
  }

  spin_lock(&vfsmount_lock);
  /* 当前场景 parent_nd 为空，跳过 */
  if (parent_nd) {
      detach_mnt(source_mnt, parent_nd);
      attach_mnt(source_mnt, nd);
      touch_namespace(current->namespace);
  } else {
      mnt_set_mountpoint(dest_mnt, dest_dentry, source_mnt);
      commit_tree(source_mnt);
  }

  list_for_each_entry_safe(child, p, &tree_list, mnt_hash) {
      list_del_init(&child->mnt_hash);
      commit_tree(child);
  }
  spin_unlock(&vfsmount_lock);
  return 0;
}
```

mnt_set_mountpoint 函数里面，目的 dentry 的 d_mounted 要加 1，这是将来判断该 dentry 是否是为挂载点的依据。同时源文件系统 vfsmout 对象的 mnt_mountpoint 指向目的 dentry。

commit_tree 用来把源 vfsmount 提交到全局 hash 链表，它的代码如代码清单 2-27 所示。

代码清单 2-27 commit_tree 函数

```
193  static void commit_tree(struct vfsmount *mnt)
194  {
195    struct vfsmount *parent = mnt->mnt_parent;
196    struct vfsmount *m;
197    LIST_HEAD(head);
198    struct namespace *n = parent->mnt_namespace;
199
200    BUG_ON(parent == mnt);
201
202    list_add_tail(&head, &mnt->mnt_list);
203    list_for_each_entry(m, &head, mnt_list)
204         m->mnt_namespace = n;
205    list_splice(&head, n->list.prev);
206    /* 源 vfsmount 对象链接到 mount hash 表 */
207    list_add_tail(&mnt->mnt_hash, mount_hashtable +
208                  hash(parent, mnt->mnt_mountpoint));
       /* 源 vfsmount 对象链接到父 vfsmount 对象的链表 */
209    list_add_tail(&mnt->mnt_child, &parent->mnt_mounts);
210    touch_namespace(n);
211  }
```

commit_tree 从 202 行到 205 行目的是把源文件系统的 vfsmount 对象链接到 namespace 的链表尾部。链接之前，第 204 行设置 vfsmount 对象的 namespace 为父对象的 namespace。

执行到最后，mount 实际上把 aufs 的 vfsmount 对象链接到一个全局 hash 表，同时也链接到父 vfsmount 对象的链表头。而目的 dentry 指向的就是目的文件系统的 au 目录。这个 dentry 结构的 d_mounted 成员要加 1，这是判断这个目录是否是挂载点的依据。在文件打开的过程需要判断目录是否挂载点，检查的依据就是这个参数。

代码分析到现在，已经大量使用了双向链表 list 和 hash list。理解这些基础的数据结构是阅读代码的基础条件。

2.3.4 文件打开的代码分析

通过前面代码的工作，源文件系统的 vfsmount 对象已经链接到目的文件系统的 vfsmount 对象，同时也链接到全局的 hash 表。

1. sys_open 函数

打开一个文件，是通过内核提供的系统调用 sys_open 实现的。我们来分析一下 sys_open 函数，它的代码如代码清单 2-28 所示。

代码清单 2-28 sys_open 函数

```
asmlinkage long sys_open(const char __user *filename, int flags, int mode)
{
  long ret;

  if (force_o_largefile())
      flags |= O_LARGEFILE;

  /*AT_FDCWD 指示文件的查找位置，后面要用到 */
     ret = do_sys_open(AT_FDCWD, filename, flags, mode);
  /* 禁止编译器的尾部调用优化 */
  /* avoid REGPARM breakage on x86: */
  prevent_tail_call(ret);
  return ret;
}
```

sys_open 是 do_sys_open 的封装函数。函数 do_sys_open 的代码如代码清单 2-29 所示。

代码清单 2-29 do_sys_open 函数

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
  char *tmp = getname(filename);
  int fd = PTR_ERR(tmp);

  if (!IS_ERR(tmp)) {
      fd = get_unused_fd();
      if (fd >= 0) {
          struct file *f = do_filp_open(dfd, tmp, flags, mode);
          if (IS_ERR(f)) {
              put_unused_fd(fd);
              fd = PTR_ERR(f);
          } else {
              fsnotify_open(f->f_dentry);
              /* 安装文件指针到 fd 数组 */
              fd_install(fd, f);
```

函数 do_sys_open 首先把文件名从用户态复制到内核，然后获得一个未使用的文件号，最后调用 do_filp_open 执行文件打开的过程。

2. do_filp_open 函数

函数 do_filp_open 代码如代码清单 2-30 所示。

代码清单 2-30 do_filp_open 函数

```
static struct file *do_filp_open(int dfd, const char *filename, int flags, int mode)
{
  int namei_flags, error;
  struct nameidata nd;
```

```
    /* 设置文件的标志，为何要重新设置？看英文注释，这是因为内部的标志和外部定义不一致 */
    namei_flags = flags;
    if ((namei_flags+1) & O_ACCMODE)
        namei_flags++;

    error = open_namei(dfd, filename, namei_flags, mode, &nd);
    if (!error)
        return nameidata_to_filp(&nd, flags);

    return ERR_PTR(error);
}
```

do_filp_open 主要执行了两步。

- 第一步是 open_namei，它的作用是沿着要打开文件名的整个路径，一层层解析路径，最后得到文件的 dentry 和 vfsmount 对象，保存到一个 nameidata 结构中。这个 nameidata 结构就是 open_namei 的输入参数 nd。
- 第二步是 nameidata_to_filp 函数，它的作用是根据第一步获得的 nameidata 结构，初始化一个 file 对象。

3. open_namei 函数

我们首先从 open_namei 函数开始分析，它的代码如代码清单 2-31 所示。

代码清单 2-31 open_namei 函数

```
int open_namei(int dfd, const char *pathname, int flag,
        int mode, struct nameidata *nd)
{
    int acc_mode, error;
    struct path path;
    struct dentry *dir;
    int count = 0;

    acc_mode = ACC_MODE(flag);

    /* 检查写权限 */
    /* O_TRUNC implies we need access checks for write permissions */
    if (flag & O_TRUNC)
        acc_mode |= MAY_WRITE;

      /* Allow the LSM permission hook to distinguish append
       access from general write access. */
    if (flag & O_APPEND)
        acc_mode |= MAY_APPEND;
```

open_namei 函数第一部分是设置权限参数。如果打开文件时带有 O_TRUNC 标志，说明要修改文件的长度，对文件的操作模式要加上可写权限；如果打开文件时带有 O_APPEND 标志，对文件的操作模式要加上 MAY_APPEND 权限。MAY_APPEND 也可当做可写权限

(MAY_WRITE) 的一种，但是把它专门选出来，作为一个特殊的标识。

```
/*
 * The simplest case - just a plain lookup.
 */
if (!(flag & O_CREAT)) {
        error = path_lookup_open(dfd, pathname, lookup_flags(flag),
                        nd, flag);
        if (error)
            return error;
        goto ok;
}

/*
 * Create - we need to know the parent.
 */
error = path_lookup_create(dfd,pathname,LOOKUP_PARENT,nd,flag,mode);
if (error)
        return error;
```

open_namei 函数第二部分是两种打开文件的模式。打开文件的时候，如果文件不存在，可以为用户创建一个文件，这是通过文件的 O_CREAT 标志来控制的。如果不带有 O_CREAT 标志，不需要创建文件，那么调用 path_lookup_open 函数。如果带有 O_CREAT 标志，说明需要创建文件，则调用 path_lookup_create 函数（调用函数时带有 LOOKUP_PARENT 标志）。

path_lookup_create 函数不处理最终目标文件，它只查找到文件所在目录就结束查找过程了，等函数返回后，再检查最终目标文件是否存在。

```
/*
 * We have the parent and last component. First of all, check
 * that we are not asked to creat(2) an obvious directory - that
 * will not do.
 */
error = -EISDIR;
   /* 检查 last_type 和文件名 */
if (nd->last_type != LAST_NORM || nd->last.name[nd->last.len])
      goto exit;

 /* 在父 dentry 里面查找是否有 nd->last 名字的文件，没有则创建 dentry 对象
  * 里面调用了 cached_lookup，这个函数前面分析过了 */
 dir = nd->dentry;
 nd->flags &= ~LOOKUP_PARENT;
 mutex_lock(&dir->d_inode->i_mutex);
 path.dentry = lookup_hash(nd);
 path.mnt = nd->mnt;
```

open_namei 函数第三部分首先检查 path_lookup_create 函数的返回值。第一种情况是检查返回类型。返回类型有很多种，可以是 LAST_NORM、LAST_DOTDOT、LAST_DOTDOT 等。如果返回类型不等于 LAST_NORM，说明文件名字是点“.”或者点点“..”，

则直接返回。

另外一种情况是文件名是一个目录，也不处理，直接返回。如果文件名是目录，那么文件名的最后一个字符是斜杠符“/”，而普通文件的最后一个字符不可能是斜杠符，因此目录文件的长度比 nd 指示的文件名长度多出来一个字符，通过检查最后一个字符是否为空判断文件是否是目录。文件名和长度的处理在本节的 __link_path_walk 函数继续分析。

如果检查通过，返回的 nd 变量的 dentry 成员是文件所在目录的 dentry，调用 lookup_hash 查找目标文件的 dentry，结果分两种情况。一种是文件存在，可以找到，一种是文件不存在，这种情况要为文件创建一个 dentry 结构。lookup_hash 函数调用了 __lookup_hash 在 dentry cache 执行查找过程。__lookup_hash 函数在 aufs 文件系统一节已经分析过，此处略过。

```
do_last:
....../* 省略无关代码 */
   /*d-inode 为空，说明文件不存在，需要创建 inode*/
  /* Negative dentry, just create the file */
  if (!path.dentry->d_inode) {
      if (!IS_POSIXACL(dir->d_inode))
          mode &= ~current->fs->umask;
      error = vfs_create(dir->d_inode, path.dentry, mode, nd);
      mutex_unlock(&dir->d_inode->i_mutex);
      dput(nd->dentry);
      nd->dentry = path.dentry;
      if (error)
          goto exit;
      /* Don't check for write permission, don't truncate */
      acc_mode = 0;
      flag &= ~O_TRUNC;
      goto ok;
  }

  /*
   * It already exists.
   */
  mutex_unlock(&dir->d_inode->i_mutex);
  audit_inode_update(path.dentry->d_inode);

  error = -EEXIST;
  if (flag & O_EXCL)
      goto exit_dput;
    /* 检查是否一个 mount 点，如果是 mount 点需要切换到源 mount 点 */
  if (__follow_mount(&path)) {
      error = -ELOOP;
      if (flag & O_NOFOLLOW)
          goto exit_dput;
  }

  error = -ENOENT;
  if (!path.dentry->d_inode)
      goto exit_dput;
```

```
        /* 是否一个符号链接，是则跳到 do_link 分支处理 */
    if (path.dentry->d_inode->i_op && path.dentry->d_inode->i_op->follow_link)
        goto do_link;

    path_to_nameidata(&path, nd);
    error = -EISDIR;
        /* 如果是目录，出错退出 */
    if (path.dentry->d_inode && S_ISDIR(path.dentry->d_inode->i_mode))
        goto exit;
ok:
        /* 最后统一的 open 处理 */
    error = may_open(nd, acc_mode, flag);
```

open_namei 函数第四部分首先检查 dentry 结构的 d_inode 成员。如果成员为空，说明文件不存在，dentry 是函数第三部分刚刚创建的，因此 d_inode 尚未赋值为空。这种情况下，调用 vfs_create 创建文件，然后进入 ok 分支返回。

如果成员不为空，说明文件已经存在，随后要检查文件为挂载点或者符号链接的情况。这两种情况在本节后面的 __link_path_walk 函数也要处理，在后面代码中一并分析。

```
do_link:
    /* 如果是符号链接，继续处理，找到符号链接的目的文件 */
    error = __do_follow_link(&path, nd);
    ......
    if (nd->last.name[nd->last.len]) {
        __putname(nd->last.name);
        goto exit;
    }
    error = -ELOOP;
    if (count++==32) {
        __putname(nd->last.name);
        goto exit;
    }
    dir = nd->dentry;
    mutex_lock(&dir->d_inode->i_mutex);
    path.dentry = lookup_hash(nd);
    path.mnt = nd->mnt;
    __putname(nd->last.name);
    goto do_last;
}
```

open_namei 函数第五部分是进行符号链接的处理。调用 __do_follow_link 为符号链接文件找到它的真实文件，然后再执行真实文件的查找过程。因为符号链接可以一层层链接，造成无限的循环，所以需要设置一个变量 count 计算符号链接的递归次数。超过设定值 32 以上的，就不再解析，而是返回错误。

4. path_lookup_create 和 _path_lookup_intent_open 函数

path_lookup_create 函数顾名思义，它是沿着文件的整个路径寻找。文件的路径是包含斜杠符“/”的一串字符，path_lookup_create 要对字符进行分割，分离出每层路径的目录名和

最终的文件名，然后对目录和最终的文件进行查找。

path_lookup_open 和 path_lookup_create 都封装了 __path_lookup_intent_open 函数，不同之处只是 path_lookup_create 的参数带有 LOOKUP_CREATE 标志，所以只分析 __path_lookup_intent_open 函数就可以了，如代码清单 2-32 所示。

代码清单 2-32 __path_lookup_intent_open 函数

```
static int __path_lookup_intent_open(int dfd, const char *name,
      unsigned int lookup_flags, struct nameidata *nd,
      int open_flags, int create_mode)
  {  /* 创建一个新的 filp 对象 */
  struct file *filp = get_empty_filp();
  int err;

  if (filp == NULL)
      return -ENFILE;
  nd->intent.open.file = filp;
  nd->intent.open.flags = open_flags;
  nd->intent.open.create_mode = create_mode;
  err = do_path_lookup(dfd, name, lookup_flags|LOOKUP_OPEN, nd);
```

5. do_path_lookup 函数

__path_lookup_intent_open 函数给 nd 参数赋值之后，调用 do_path_lookup 执行路径查找工作，它的代码如代码清单 2-33 所示。

代码清单 2-33 do_path_lookup 函数

```
/* Returns 0 and nd will be valid on success; Retuns error, otherwise. */
static int fastcall do_path_lookup(int dfd, const char *name,
               unsigned int flags, struct nameidata *nd)
{
……/* 省略参数定义代码 */
  nd->last_type = LAST_ROOT; /* if there are only slashes... */
  nd->flags = flags;
  nd->depth = 0;

  if (*name=='/') {
      read_lock(&current->fs->lock);
      if (current->fs->altroot && !(nd->flags & LOOKUP_NOALT)) {
          /**/
           nd->mnt = mntget(current->fs->altrootmnt);
           nd->dentry = dget(current->fs->altroot);
           read_unlock(&current->fs->lock);
           if (__emul_lookup_dentry(name,nd))
               goto out; /* found in altroot */
           read_lock(&current->fs->lock);
      }
      /* 查找的 dentry 对象和 vfsmount 对象是文件系统的 root dentry 和 root vfsmount*/
      nd->mnt = mntget(current->fs->rootmnt);
```

```
        nd->dentry = dget(current->fs->root);
        read_unlock(&current->fs->lock);
```

do_path_lookup 函数的第一部分是检查文件名字是否用斜杆符“/”开始，以斜杠符开始，说明文件的查找是从根目录开始，那么起始的 dentry，也就是 nd 变量的 dentry 要设置为当前进程文件系统的根 dentry，nd 变量的 vfsmount 对象要设置为当前进程文件系统的根 vfsmount 对象。当前进程文件系统是一个和进程有关的概念，每个进程初始化的时候，都要为它设置当前文件系统。当前文件系统包含了三个 dentry，它们分别指向根 dentry、当前 dentry（即 pwd 命令显示的当前目录）和替换根 dentry。

如果当前进程文件系统存在替换根 dentry 且打开文件的时候不设置 LOOKUP_NOALT 标志，那么 nd 变量的 dentry 要设置为当前进程文件系统的替换根 dentry，nd 变量的 vfsmount 对象要设置为当前进程文件系统的替换根 vfsmount 对象。

```
    } else if (dfd == AT_FDCWD) {
            /* 如果是 AT_FDCWD，说明要在进程的当前路径查找文件，那么 dentry 就是进程当前的 dentry 对象 */
            read_lock(&current->fs->lock);
            nd->mnt = mntget(current->fs->pwdmnt);
            nd->dentry = dget(current->fs->pwd);
            read_unlock(&current->fs->lock);
```

do_path_lookup 第二部分检查 AT_FDCWD 标志。如果文件名不是用斜杠符开始而且设置了 AT_FDCWD 标志，意味着文件的搜索路径不是从进程文件系统的根路径开始，而是从进程当前路径开始。所以设置 nd 变量的 dentry 为进程文件系统的当前 dentry，nd 变量的 vfsmount 对象要设置为进程文件系统的当前 vfsmount 对象。

```
    } else {
          struct dentry *dentry;

          file = fget_light(dfd, &fput_needed);
          retval = -EBADF;
          if (!file)
              goto out_fail;

          dentry = file->f_dentry;
          ......
          nd->mnt = mntget(file->f_vfsmnt);
          nd->dentry = dget(dentry);

          fput_light(file, fput_needed);
      }
      current->total_link_count = 0;
      retval = link_path_walk(name, nd);
      ......
    }
```

do_path_lookup 第三部分的前提是前两个部分的条件都不成立，这个文件已经打开过，输入参数是文件的 ID 号。这种情况是在进程的已打开文件结构里面根据用户态的 ID 号查找

文件。这种场景不是我们研究的情况，此处略过。

6. link_path_walk 函数

当 nd 参数设置了正确的 dentry 和 vfsmount 对象后，调用 link_path_walk 执行路径的查找。link_path_walk 执行了两次查找的过程，如代码清单 2-34 所示。

代码清单 2-34 link_path_walk 函数

```
int fastcall link_path_walk(const char *name, struct nameidata *nd)
{
  struct nameidata save = *nd;
  int result;

  /* make sure the stuff we saved doesn't go away */
  /* 增加 mnt 和 dentry 的引用计数 */
  dget(save.dentry);
  mntget(save.mnt);

  result = __link_path_walk(name, nd);
  if (result == -ESTALE) {
      *nd = save;
      dget(nd->dentry);
      mntget(nd->mnt);
      nd->flags |= LOOKUP_REVAL;
      result = __link_path_walk(name, nd);
  }

  dput(save.dentry);
  mntput(save.mnt);

  return result;
}
```

link_path_walk 两次执行了 __link_path_walk 函数。原因是第一次查找有可能失败，文件系统返回了 ESTALE 失败标识码。这种情况下 nd 的 flag 成员要加上 LOOKUP_REVAL 标志，作用是不再依赖 dentry cache，而是强迫文件系统执行自己的查找功能。对于硬盘文件系统，文件系统自己的查找功能通常要读硬盘上文件系统的元数据来获取文件信息。

7. __link_path_walk 函数

__link_path_walk 真正对文件的整个路径名做循环查找，需要一层层解析文件的路径，对每层路径进行查询，如代码清单 2-35 所示。

代码清单 2-35 __link_path_walk 函数

```
static fastcall int __link_path_walk(const char * name, struct nameidata *nd)
{
  struct path next;
  struct inode *inode;
  int err;
```

```
    unsigned int lookup_flags = nd->flags;

    while (*name=='/')
        name++;
    if (!*name)
        goto return_reval;

    inode = nd->dentry->d_inode;
    if (nd->depth)
        lookup_flags = LOOKUP_FOLLOW | (nd->flags & LOOKUP_CONTINUE);
```

__link_path_walk 函数非常复杂，我们把它分为多个部分，逐一讲解。

第一部分是将文件名字符串的最前面的斜杠符去掉。因为斜杠符可能是多个，所以有一个 while 循环。然后检查文件符号链接的深度。因为文件可以是一个符号链接，符号链接又可以指向一个符号链接，如此递归可能造成死循环。每次处理符号链接的时候，结构 nd 的 depth 成员加一，如果超过一个限值，就不再处理了，避免无限的符号链接。

```
/* At this point we know we have a real path component. */
/* 这个循环遍历名字字符的每一轮。就是以“/”字符分隔的每一层字符 */
for(;;) {
    unsigned long hash;
    struct qstr this;
    unsigned int c;
    nd->flags |= LOOKUP_CONTINUE;
    /* 权限检查 */
    err = exec_permission_lite(inode, nd);
    if (err == -EAGAIN)
        err = vfs_permission(nd, MAY_EXEC);
    if (err)
        break;

    /* 这里计算 name 的 hash 值，如果碰到“/”字符，代表这一轮的名字到了结束位置 */
    this.name = name;
    c = *(const unsigned char *)name;

    hash = init_name_hash();
    do {
        name++;
        hash = partial_name_hash(c, hash);
        c = *(const unsigned char *)name;
    } while (c && (c != '/'));
    this.len = name - (const char *) this.name;
    this.hash = end_name_hash(hash);

    /* remove trailing slashes? */
    /* 已经是最后一轮的名字字符了，转到 last_component 处理 */
    if (!c)
        goto last_component;
    /* 最后的字符是个“/”，转到 last_with_slashes 处理 */
```

```
    while (*++name == '/');
    if (!*name)
        goto last_with_slashes;
```

__link_path_walk 函数第二部分是一个循环，作用是将文件名字符串分离。文件名字符串是一个长的路径，每一层目录之间用斜杠符分隔。这部分代码逐个比较字符，如果碰到了斜杠符，意味着斜杠符之前的字符串是一个目录名。对目录名计算 hash 值，之后进行目录名的查找工作。计算 hash 值的过程在 aufs 文件系统的例子中已经分析过，不再赘述。如果目录名查找成功，则进入下一轮循环。

循环最终有两种情况，一种情况是得到了最终的目标文件名，转入 last_component 分支处理，另一种是整个文件名字符用的是一个斜杠符结尾的，则转入 last_with_slashes 分支处理。这两个分支的名字其实就说明了它们各自的功能。

```
/*
 * "." and ".." are special - ".." especially so because it has
 * to be able to know about the current root directory and
 * parent relationships.
 */
/* 如果名字是“.”或“..”，要特殊处理 */
if (this.name[0] == '.') switch (this.len) {
    default:
        break;
    case 2:
        if (this.name[1] != '.')
            break;
        follow_dotdot(nd);
        inode = nd->dentry->d_inode;
        /* fallthrough */
    case 1:
        continue;
}
/* 如果文件系统提供了自己的 hash 函数，则使用它计算 hash 值 */
if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
    err = nd->dentry->d_op->d_hash(nd->dentry, &this);
    if (err < 0)
        break;
}
```

__link_path_walk 函数第三部分是处理文件名中特殊的点（“.”）和点点（“..”）字符。文件名是一个点代表自身，文件名是两个点代表上一级目录。所以文件名是一个点的时候什么也不做，直接进入下一级循环，如果文件名是两个点，则调用 follow_dotdot 寻找当前文件的上一级目录。

```
/* This does the actual lookups.. */
err = do_lookup(nd, &this, &next);
if (err)
    break;
```

```
err = -ENOENT;
/*d_inode 是所查找到文件的 inode，如果为空，说明查找失败 */
inode = next.dentry->d_inode;
if (!inode)
    goto out_dput;
err = -ENOTDIR;
if (!inode->i_op)
    goto out_dput;
/*inode 有 follow_link，说明是个符号链接，要特殊处理，否则执行 path_to_nameidata*/
if (inode->i_op->follow_link) {
    err = do_follow_link(&next, nd);
    if (err)
        goto return_err;
    err = -ENOENT;
    inode = nd->dentry->d_inode;
    if (!inode)
        break;
    err = -ENOTDIR;
    if (!inode->i_op)
        break;
} else
    path_to_nameidata(&next, nd);
err = -ENOTDIR;
if (!inode->i_op->lookup)
    break;
continue;
/* here ends the main loop */
```

__link_path_walk 函数第四部分调用 do_lookup 函数执行真正的查找，查找的结果通过一个 path 结构变量 next 返回。

do_lookup 执行的是最终目标文件的目录的查找，最终的目标文件不是它查找，而是在 last_component 分支中执行的。所以如果 inode 不具备 i_op 成员（目录必须有该成员），说明该 inode 不是一个目录，则返回 ENOTDIR 错误。这个错误的英文名字就说明了错误原因。

如果 i_op 成员具备 follow_link 成员，说明 inode 是一个符号链接。符号链接必须调用 do_follow_link 找到符号链接真正指向的路径。

```
last_with_slashes:
      /* 最后是个“/”，说明是个目录，设置标志 */
      lookup_flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
last_component:
      /* 已经沿文件路径到了最后，就是说找到了文件本身 */
      /* Clear LOOKUP_CONTINUE iff it was previously unset */
      nd->flags &= lookup_flags | ~LOOKUP_CONTINUE;
      if (lookup_flags & LOOKUP_PARENT)
          goto lookup_parent;

      /* 最后的文件名字是“.”或者“..”，要特殊处理 */
      if (this.name[0] == '.') switch (this.len) {
          default:
```

```
            break;
        case 2:
            if (this.name[1] != '.')
                break;
            follow_dotdot(nd);
            inode = nd->dentry->d_inode;
                /* fallthrough */
        case 1:
            goto return_reval;
    }
    if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
        err = nd->dentry->d_op->d_hash(nd->dentry, &this);
        if (err < 0)
            break;
    }
    /* 查找最终的文件 */
    err = do_lookup(nd, &this, &next);
    if (err)
        break;
    inode = next.dentry->d_inode;
    /* 最终文件是个符号链接，要特殊处理 */
    if ((lookup_flags & LOOKUP_FOLLOW)
        && inode && inode->i_op && inode->i_op->follow_link) {
         err = do_follow_link(&next, nd);
         if (err)
             goto return_err;
         inode = nd->dentry->d_inode;
    } else
        path_to_nameidata(&next, nd);
    err = -ENOENT;
    if (!inode)
        break;
    /* 带有 LOOKUP_DIRECTORY 的标志，说明打开的是目录。非我们研究的情况，跳过 */
    if (lookup_flags & LOOKUP_DIRECTORY) {
        err = -ENOTDIR;
        if (!inode->i_op || !inode->i_op->lookup)
            break;
    }
    goto return_base;
```

__link_path_walk 函数第五部分是 last_with_slashes 分支和 last_component 分支。对于 last_with_slashes 分支要加上一个 LOOKUP_DIRECTORY 标志。意味着最终查找的是一个目录。

如果参数带有 LOOKUP_PARENT 标志，进入 lookup_parent 分支处理，而不在 last_component 分支处理。这个标志是目标文件有可能不存在需要创建的时候使用，这说明 last_component 只处理最终目标文件存在的情况。如果有可能不存在，则需要 lookup_parent 分支处理。

last_component 分支同样要处理文件名是“点”和“点点”的情况，这种情况在前面已

经分析过。

对最终的目标文件，last_component 分支同样调用 do_lookup 执行最终目标文件的查找，对查找的结果，也要处理符号链接的情况。这种情况和第四部分重合。

```
lookup_parent:
        /*last 成员返回最终目标文件的名字和长度 */
        nd->last = this;
        nd->last_type = LAST_NORM;
        if (this.name[0] != '.')
            goto return_base;
        if (this.len == 1)
            nd->last_type = LAST_DOT;
        else if (this.len == 2 && this.name[1] == '.')
            nd->last_type = LAST_DOTDOT;
        else
            goto return_base;
```

__link_path_walk 函数第六部分是 lookup_parent 分支，这个分支专门处理最终目标文件有可能不存在的场景。这个分支设置 nd 的返回类型是 LAST_NORM 就直接返回了，并没有真正去执行查找。这种情况最终目标文件的查找是在 open_namei 函数执行的，前面已经分析过。如果最终的目标文件名是点（“.”）或者点点（“..”），返回的 last_type 要表明是 LAST_DOT 或者 LAST_DOTDOT。

__link_path_walk 函数很复杂，为了更清晰的理解，我们借助一个例子来分析。假设把 aufs 文件系统挂载到了 /home/mnt/au，我们要打开的文件是 home/mnt/au/woman star/lbb。具体步骤如下。

步骤 1 首先是根据分隔号得到 home 目录，然后计算 home 的 hash 值，调用 do_lookup。

步骤 2 这样就获得了 home 文件（目录文件）的 inode，因为 home 的 inode 无 follow_link 调用，最终调用 path_to_nameidata。

步骤 3 对 mnt 目录用步骤 1 和步骤 2 查找。

步骤 4 查找 au。因为 au 是个挂载点，在 do_lookup 函数需要根据两个文件系统的挂载点解析，解析后，父目录就换成了 aufs 的根目录。这部分下一节再分析。

步骤 5 对 woman star 目录用步骤 1 和步骤 2 查找。

步骤 6 最后的目标文件 lbb 由 last_component 分支处理。在 woman star 目录可以找到。

通过例子，应该可以清楚理解 __link_path_walk 函数的处理流程。对于路径名中间出现的点“.”或者点点“..”，以及符号链接的处理，读者可以自行分析一下。

8. do_lookup 函数

do_lookup 函数不仅执行最终目标文件的查找，还要处理挂载点，它的代码如代码清单 2-36 所示。

代码清单 2-36 do_lookup 函数

```
static int do_lookup(struct nameidata *nd, struct qstr *name,
            struct path *path)
```

```
{
  struct vfsmount *mnt = nd->mnt;
  struct dentry *dentry = __d_lookup(nd->dentry, name);

  if (!dentry)
      goto need_lookup;
  if (dentry->d_op && dentry->d_op->d_revalidate)
      goto need_revalidate;
```

do_lookup 函数第一部分以 nd 的 dentry 为父目录，调用 __d_lookup 函数查找 name 所代表文件的 dentry。name 代表的文件既可能是一个普通文件，也可能是一个目录文件。

do_lookup 函数第二部分是两个分支。一个是 done 分支，另一个是 need_lookup 分支。

```
done:
  path->mnt = mnt;
  path->dentry = dentry;
  __follow_mount(path);
  return 0;

need_lookup:
  /*cache 找不到，需要真正查找。这是文件系统提供的 lookup 调用 */
  dentry = real_lookup(nd->dentry, name, nd);
  if (IS_ERR(dentry))
      goto fail;
  goto done;
```

如果第一部分的查找成功了，则进入 done 分支设置 vfsmount 对象和 dentry，然后检查 dentry 是否一个挂载点。如果查找未成功，则进入 need_lookup 分支。第一部分的查找是在 dentry cache 里面进行，如果进入 need_lookup 分支，说明在 dentry cache 中找不到指定名字文件的 dentry。对于建立在硬盘之上的文件系统，这时候要调用文件系统提供的 lookup 函数在硬盘上搜索文件。这部分代码分析深入的话，就涉及块设备读写和文件的读写，后文再分析。对于我们的情况，不需要调用文件系统的 lookup 函数。完成文件系统的 lookup 之后，仍然需要进入 done 分支，处理挂载点。

9. __follow_mount 函数

挂载点的处理需要调用 __follow_mount 函数，找到挂载点真正的 dentry 结构，如代码清单 2-37 所示。

代码清单 2-37 __follow_mount 函数

```
static int __follow_mount(struct path *path)
{
  int res = 0;
  while (d_mountpoint(path->dentry)) {
      /* 查找挂载的对象 */
      struct vfsmount *mounted = lookup_mnt(path->mnt, path->dentry);
      if (!mounted)
          break;
```

```
        dput(path->dentry);
        if (res)
            mntput(path->mnt);
        /* 找到挂载点，更换 dentry 为挂载点的 root dentry*/
        path->mnt = mounted;
        path->dentry = dget(mounted->mnt_root);
        res = 1;
    }
    return res;
}
```

d_mountpoint 函数用于判断该 dentry 是否是挂载点，也就是判断 d_mounted 参数是否为 0。此时回顾 mount 系统调用。当一个文件系统挂载的时候，这个参数要加 1，所以如果该文件的 dentry 被一个文件系统挂载了，这个参数不为 0。

lookup_mnt 是遍历系统的 mount 链表，找到挂载点，然后更换 dentry。还是用例子来解释。因为 au 是个挂载点，所以 lookup_mnt 的参数就是 au 的 vfsmount 对象和 dentry。aufs 文件系统挂载时，已经把文件系统的 vfsmount 对象链接到 mount 链表，lookup_mnt 找到的结果就是 aufs 文件系统的 vfsmount 对象。那么 path 的 dentry 就换成了 aufs 文件系统的 root dentry。当 do_lookup 函数返回后，下一轮要寻找 woman star 目录，实际是在 aufs 的根目录里面查找。

现在总结 open_namei 的整个处理过程：经过层层解析，open_namei 函数最终结果是得到了文件的 dentry 和 inode 结构，以及 vfsmount 对象。

10. nameidata_to_filp 函数

从当前代码返回 do_filp_open 函数。open_namei 之后，要调用 nameidata_to_filp 函数，实现打开文件的最后一步，获得文件结构，它的代码如代码清单 2-38 所示。

代码清单 2-38　nameidata_to_filp 函数

```
struct file *nameidata_to_filp(struct nameidata *nd, int flags)
{
  struct file *filp;

  /* Pick up the filp from the open intent */
  filp = nd->intent.open.file;
  /* Has the filesystem initialised the file for us? */
      /* 如果文件系统没有初始化 f_dentry*/
  if (filp->f_dentry == NULL)
      filp = __dentry_open(nd->dentry, nd->mnt, flags, filp, NULL);
  else
      path_release(nd);
  return filp;
}
```

11. __dentry_open 函数

文件结构 filp 已经在 open_namei 的过程中创建了，只不过它还没完成初始化。

对它初始化通过 __dentry_open 函数执行，初始化过程要对文件打开时设置的选项进行处理，如代码清单 2-39 所示。

代码清单 2-39 __dentry_open 函数

```
static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
                    int flags, struct file *f,
                    int (*open)(struct inode *, struct file *))
{
  struct inode *inode;
  int error;

  f->f_flags = flags;
  f->f_mode = ((flags+1) & O_ACCMODE) | FMODE_LSEEK |
              FMODE_PREAD | FMODE_PWRITE;
  inode = dentry->d_inode;
  /* 如果允许写文件，检查写权限 */
  if (f->f_mode & FMODE_WRITE) {
      error = get_write_access(inode);
      if (error)
          goto cleanup_file;
  }
  /* 给文件的参数赋值 */
  f->f_mapping = inode->i_mapping;
  f->f_dentry = dentry;
  f->f_vfsmnt = mnt;
  f->f_pos = 0;
  f->f_op = fops_get(inode->i_fop);
  file_move(f, &inode->i_sb->s_files);
```

__dentry_open 函数第一部分主要是初始化文件的参数。文件的操作函数 f_op 从 inode 获得。f_mapping 是文件的读写 cache 的管理结构，同样从 inode 获得。函数 file_move 把文件加入超级块对象的文件链表，这样从超级块可以遍历文件系统内所有的文件结构。

```
if (!open && f->f_op)
    open = f->f_op->open;
if (open) {
    error = open(inode, f);
    if (error)
        goto cleanup_all;
}

f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
/* 初始化文件的预读参数 */
file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);

/* NB: we're sure to have correct a_ops only after f_op->open */
```

```
/* 如果文件带有O_DIRECT标志，检查direct I/O的函数调用 */
if (f->f_flags & O_DIRECT) {
    if (!f->f_mapping->a_ops ||
        ((!f->f_mapping->a_ops->direct_IO) &&
        (!f->f_mapping->a_ops->get_xip_page))) {
        fput(f);
        f = ERR_PTR(-EINVAL);
    }
}
```

__dentry_open 函数第二部分首先检查文件系统是否为文件定义了 open 函数，如果已经定义，那么随后执行文件的 open 函数。然后函数 file_ra_state_init 初始化文件预读的参数。

文件预读相关的内容在第 10 章。最后是处理 O_DIRECT 模式，这是通过 direct I/O 方式访问文件，不经过文件的 page cache，在第 10 章读写文件的流程将看到它的作用。

2.4 本章小结

本章通过一个简单的文件系统，分析了文件系统挂载、文件和目录的创建，以及文件打开的过程。通过这些分析，读者对文件系统的概念、超级块、inode、dentry 的概念，以及架构应该有比较深入的理解。借助这些知识，完全可以分析文件关闭的过程，或者 chmod、ustat、utime、truncate 等文件系统调用的实现。

第 3 章

设备的概念和总体架构

CPU、内存和设备是计算机最重要的三个物质基础。对设备的理解，也是我们理解驱动架构，总线架构的基础。

通常的显卡网卡声卡等设备，都是先插入计算机系统的 PCI 总线插槽（早期还有 ISA、MCA 总线等。现在 PC 领域基本 PCI 总线统一天下），安装驱动之后，应用程序可以通过文件系统打开和读写设备文件。这个过程可以从三个层面理解：1）设备本身的特性；2）总线和操作系统对设备的管理；3）设备的驱动层。其中，后两个层面将在本书第 8 章重点分析，第 6 章和第 7 章也有很多内容涉及这两个层面。设备的特性这个层面是理解设备的基础，也是正确理解其他层面的基础，本章重点介绍设备的特性，对总线和操作系统对设备的管理以及设备驱动只做简单描述，细节部分放到后面章节。

3.1 设备的配置表

因为 PCI 设备是当前最广泛、最流行的设备，因此本章以 PCI 设备为准。以 PCI 设备为例，它本身就包含一个配置表。用图 3-1 来解释设备的配置表。

配置表包含设备制造商填充的厂商信息。设备属性等通用配置信息。此外，设备厂商还应该提供设备的控制寄存器信息，通过这些控制寄存器，系统可以设置设备的状态、控制设备的运行，或者从设备获得信息。另外，设备还可能配备了内存（有的设备可能没有），系统可以读写设备的内存。

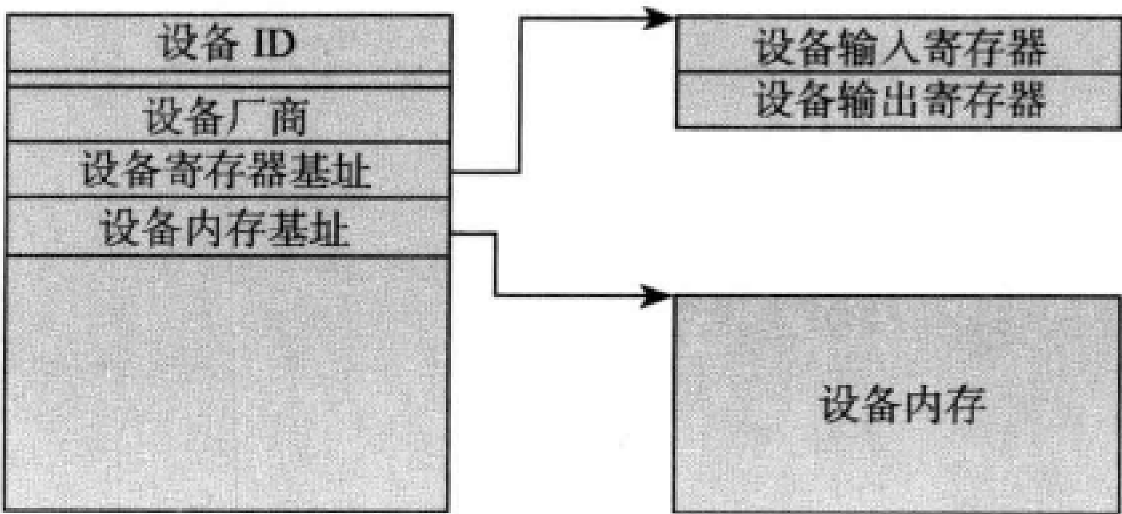


图 3-1　设备配置表的信息

设备本身有一些配置信息，如设备的 ID、制造商 ID 等。

设备内存基址，指示了设备内存的地址和长度，而设备寄存器基址，则指示了设备的寄存器地址和长度。这个设备有两个寄存器，一个输入寄存器，另一个输出寄存器。当输入寄存器写入数值后，可以

从输出寄存器读到另一个数值。

设备寄存器基址，这个概念有点难。实际上，可以将其看做一个地址，对这个地址写指令就可以控制设备。所以，设备寄存器其实就是设备的控制接口。

PCI 总线规范定义的 PCI 设备配置空间总长度为 256 字节，配置信息按一定的顺序和大小依次存放。配置空间的前 64 字节称为**配置头**。对于所有的设备而言，配置头的主要功能是用来识别设备、定义主机访问 PCI 卡的方式。其余 192 字节称为**本地配置空间**，主要定义卡上局部总线的特性、本地空间基地址及范围等。

3.2 访问设备寄存器和设备内存

x86 系统为控制设备设置了一个地址空间，这个空间称为计算机的 I/O 端口空间，这个空间占据了 65536 个 8 位的范围。

不同的处理器对设备控制接口有不同的访问方式。对 x86 系统来说，专门提供了特别的指令来访问设备寄存器。这就是 x86 系统的 I/O 指令。

对上文的例子设备而言，需要把设备的寄存器基址纳入到系统的 I/O 端口空间里面，然后就可以通过系统提供的 I/O 指令来访问设备的寄存器。假设设备厂商提供的寄存器基址是 0x1c00，长度是 8 字节。有两种情况：

一种是 0x1c00 地址和别的设备没有冲突，可以直接使用，操作系统内核就记录设备的寄存器基址为 0x1c00，驱动通过 x86 系统提供的 I/O 指令访问 I/O 地址 0x1c00，或者叫 0x1c00 I/O 端口，就可以设置设备输入寄存器的内容。通过 I/O 指令访问地址 0x1c04，就可以读到设备输出寄存器的内容。

另外一种情况是其他设备使用了 0x1c00 这个 I/O 地址。操作系统内核就需要寻找一个合适的寄存器基址，然后更新设备的寄存器基址，并记录到内核的设备信息里面。驱动使用 x86 的 I/O 指令访问这个更新后的地址，就可以设置设备输入寄存器的内容了。

通过设备的 I/O 端口控制设备，这就是设备驱动的功能。设备厂商会提供设备寄存器的详细内容，这也是驱动开发者所必须关注的。而发现设备、扫描设备信息、为设备提供合适的 I/O 地址空间，这是内核的总线部分要处理的事情。

访问设备的内存和前面的过程有所不同。因为设备内存不占用 I/O 端口空间，而是和系统内存占据一样的地址空间。内核读取设备内存基址，然后需要找到合适的内存空间，把设备的内存映射到内存空间。这样驱动就可以用标准的内存接口访问设备的内存了。

3.3 设备中断和 DMA

设备是控制输入输出的。接收到输入信息后，如何通知主机 CPU？通常情况下，通过中断来实现（CPU 也可以轮询来检查设备），每个设备都有自己的中断号（设备可以有多个中断），对于 PCI 设备而言，中断相关信息保存在设备的配置空间里。

主机的 CPU 能访问设备的内存，那么设备能否访问系统的内存？这是可以的。设备本

身是挂载在 PCI 总线上的，设备使用的内存地址就是 PCI 总线可以访问的地址，称为**总线地址**。在 x86 系统中，总线地址和内存物理地址相同，设备直接使用物理地址访问系统内存。这种方式叫做 DMA（Direct Memory Access，直接获取内存）。

设备要通过 DMA 方式访问系统内存，就必须知道内存的总线地址。如何把内存的总线地址传送给设备？从设备的配置表可以发现，设备的寄存器里面有一个是保存 DMA 地址的，驱动设置这个寄存器的内容，然后设备就能根据该地址启动 DMA，访问主机的内存。

3.4 总线对设备的扫描

设备的配置信息提供了设备的信息和设备寄存器基址以及设备内存基址。因此首先要读到这些信息，然后操作系统才能探测到设备，理解设备的类型和型号，为设备安排正确的驱动，并为设备安排合适的 I/O 端口和 I/O 内存。

但是如何读取设备的配置信息？ PCI 总线对这个问题的解决方法是：保留 8 字节的 I/O 端口地址，就是 0xCF8 ~ 0xCFF。要访问设备的配置信息，先往 0xCF8 地址写入目标地址信息，然后通过 0xCFC 地址读数据，就可以获得这个配置信息。这里的写和读，都是使用 x86 所特有的 I/O 指令。

写入 0xCF8 的目标地址信息，包括总线号、设备号、功能号和配置寄存器地址等综合信息。当 PCI 总线读取到设备信息，系统要为设备创建一个 PCI 设备对象，设备就这样被 PCI 总线扫描进来。这个过程在第 8 章将详细分析。

3.5 设备驱动管理

完成对设备的扫描后，接下来要为设备安装正确的驱动。设备对象创建后，要把设备注册到总线。当设备注册到总线时，总要扫描一遍总线，看是否能为设备找到驱动。设备的配置表里包含了设备的厂商信息、设备型号和类型，而设备的驱动也要包含设备的型号和类型信息，如果两者匹配，说明驱动是正确的，可以为这个设备服务。

当驱动注册到总线的时候，也要扫描一遍总线，看能否找到适合该驱动的设备。扫描的方式和设备注册扫描的方式一样。

3.6 本章小结

设备是很重要的概念，也是准确理解驱动、总线等概念的基础。初次接触概念理解上难免有困难，但是不要紧，我们需要在具体的流程中逐渐深化，通过细节来真正掌握概念。

第 4 章

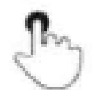
为设备服务的特殊文件系统 sysfs

sysfs 是 Linux 系统提供的一个特殊文件系统。这个文件系统的主要作用是在用户态展示设备的信息。一个安装 Linux 的计算机系统中，可以在根目录下面找到 sys 目录，这个目录实际上就是利用 sysfs 文件系统创建的。打开 sys 目录，可以看到对设备的分类显示，如图 4-1 所示。

```
[root@centos62b sys]# ls
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module  power
```

图 4-1　设备的分类

操作系统通常把设备分类为 block、bus、class、dev、devices、firmware、fs 等目录。很多读者都了解，Linux 系统是通过 proc 文件系统来管理内核的重要数据，但是随着 sysfs 文件系统越来越重要，有些内核的重要数据也开始通过 sysfs 文件系统来提供，而 proc 文件系统由于本身的缺陷，越来越少被用到。



注意

对 sys 目录进行操作可以发现，在 sys 目录下不能创建和删除文件，这是因为 sysfs 文件系统没有提供创建和删除文件的功能。和设备有关的另一个目录是 dev 目录。后面我们将看到，根目录下的 /dev 目录的设备文件只是个代表符号，不包括设备相关的信息。

4.1　文件和目录的创建

第 2 章给出了一个最简单的文件系统 aufs。本节利用 aufs 文件系统中学到的知识，继续对 sysfs 文件系统进行分析。基于已知来学习未知，可以保证知识点的衔接，同时每次学习的知识点不会太多，以防止造成理解的困难。

4.1.1 sysfs 文件系统的初始化

sysfs 本身比较简单，直接从它的初始化代码开始分析。sysfs 文件系统提供了一个初始化函数 sysfs_init 来完成注册和初始化的工作，如代码清单 4-1 所示。

代码清单 4-1 sysfs 的初始化

```
int __init sysfs_init(void)
{
  int err = -ENOMEM;

  sysfs_dir_cachep = kmem_cache_create("sysfs_dir_cache",
                          sizeof(struct sysfs_dirent),
                          0, 0, NULL, NULL);
  if (!sysfs_dir_cachep)
      goto out;

  err = register_filesystem(&sysfs_fs_type);
  if (!err) {
      sysfs_mount = kern_mount(&sysfs_fs_type);
```

sysfs_init 的代码中，register_filesystem 和 kern_mount 前文已经分析过。这两个函数的作用是把 sysfs 文件系统插入到文件系统的总链表中，然后为 sysfs 文件系统创建 vfsmount 对象、根 dentry 和根 inode 结构。

而 kmem_cache_create 函数的作用是创建一个 memory cache 对象。在第 1 章内核基础层一节分析过，它创建了一个 slab 对象，同时指定了对象的大小，以后可以利用这个对象申请内存。

sysfs 文件系统提供了函数 sysfs_get_sb，它的功能是创建文件系统超级块对象。sysfs_get_sb 函数的实现和 aufs 文件系统一样，通过调用内核提供的 get_sb_single 创建超级块对象。sysfs 调用 get_sb_single 时，提供了 sysfs_fill_super 函数作为 sysfs 文件系统超级块的赋值函数。这个赋值函数和 aufs 的赋值函数很相似，留给读者自行分析。

4.1.2 sysfs 文件系统目录的创建

对于一个文件系统，我们最关心的是文件和目录的创建和删除，以及读写。本节先介绍目录文件的创建。

1. 调用 sysfs_create_dir 函数创建目录文件

sysfs 文件系统使用 sysfs_create_dir 函数创建目录文件，其实现如代码清单 4-2 所示。

代码清单 4-2 sysfs_create_dir 函数

```
int sysfs_create_dir(struct kobject * kobj)
{
  struct dentry * dentry = NULL;
  struct dentry * parent;
```

```
    int error = 0;

    BUG_ON(!kobj);
    /* 设置父 dentry，如果没有父 dentry，指定文件系统的 root dentry 为父 dentry*/
    if (kobj->parent)
        parent = kobj->parent->dentry;
    else if (sysfs_mount && sysfs_mount->mnt_sb)
        parent = sysfs_mount->mnt_sb->s_root;
    else
        return -EFAULT;

    error = create_dir(kobj,parent,kobject_name(kobj),&dentry);
    if (!error)
        kobj->dentry = dentry;
    return error;
}
```

sysfs_create_dir 的输入参数是一个 kobject 指针。结构 kobject 和 sysfs 文件系统结合紧密，它的成员包含一个 dentry 指针。从第 2 章分析的知识点我们了解到，dentry 代表着文件系统内部的层次关系，而包含 dentry 指针的结构 kobject 可以对应到 sysfs 文件系统的一个目录，这个 dentry 指针就是目录文件的 dentry。

2. 调用 create_dir 实际执行目录的创建

sysfs_create_dir 调用 create_dir 实际执行目录的创建，它的代码如代码清单 4-3 所示。

代码清单 4-3 调用 create_dir 实际执行目录的创建

```
     static int create_dir(struct kobject * k, struct dentry * p,
124              const char * n, struct dentry ** d)
125  {
126  int error;
     /* 指定是一个目录操作 */
127  umode_t mode = S_IFDIR| S_IRWXU | S_IRUGO | S_IXUGO;
128
129  mutex_lock(&p->d_inode->i_mutex);
130  *d = lookup_one_len(n, p, strlen(n));
131  if (!IS_ERR(*d)) {
         /* 如果 dirent 对象存在，退出返回错误，否则创建一个新的 dirent*/
132      if (sysfs_dirent_exist(p->d_fsdata, n))
133          error = -EEXIST;
134      else
135          error = sysfs_make_dirent(p->d_fsdata, *d, k, mode,
136                                    SYSFS_DIR);
```

create_dir 函数的第一部分是调用 lookup_one_len 在 dentry cache 里面查找同名的 dentry。如果没有，则创建一个新的 dentry。lookup_one_len 函数前文已经分析过。

第 135 行引出一个新的概念：sysfs_dirent 结构。对 sysfs 文件系统内的每个目录和文件，都要为之创建一个 sysfs_dirent 对象。实际上，sysfs 文件系统的树形结构是通过 sysfs_dirent

保存的；而目录和文件的名字也是通过 sysfs_dirent 保存的。可以说，结构 sysfs_dirent 扮演了很重要的角色。

```
137    if (!error) {
138        error = sysfs_create(*d, mode, init_dir);
139        if (!error) {
140            p->d_inode->i_nlink++;
141            (*d)->d_op = &sysfs_dentry_ops;
142            d_rehash(*d);
143        }
144    }
```

create_dir 函数的第二部分首先调用 sysfs_create 函数为文件创建 inode 结构。如果一切正常成功，调用 d_rehash 函数把第一部分新创建的 dentry 对象链接到 dentry cache 的一个 hash 链表。dentry cache 的 hash 链表已经分析过，此处略过。

3. 调用 sysfs_make_dirent 函数创建一个 sysfs_dirent 结构

create_dir 函数调用 sysfs_make_dirent 函数创建一个 sysfs_dirent 结构，如代码清单 4-4 所示。

代码清单 4-4 调用 sysfs_make_dirent 函数创建一个 sysfs_dirent 结构

```
 int sysfs_make_dirent(struct sysfs_dirent * parent_sd, struct dentry * dentry,
           void * element, umode_t mode, int type)
{
  struct sysfs_dirent * sd;
  sd = sysfs_new_dirent(parent_sd, element);
  if (!sd)
      return -ENOMEM;

  sd->s_mode = mode;
  sd->s_type = type;
  /* 保存目录或者文件的 dentry 对象 */
      sd->s_dentry = dentry;
  if (dentry) {
      /*d_fsdata 保存 sysfs_dirent 的指针 */
      dentry->d_fsdata = sysfs_get(sd);
      dentry->d_op = &sysfs_dentry_ops;
  }

  return 0;
}
```

sysfs_make_dirent 函数调用 sysfs_new_dirent 创建一个新的 sysfs_dirent 结构，这个新的结构要链接到父结构的链表，而且保存传递进来的 element 参数。

4. 调用 sysfs_create 函数为目录文件创建一个 inode 对象

现在返回 create_dir 函数，在获得 dentry 结构之后，还需要为目录文件创建一个 inode 对象，这是通过 sysfs_create 函数实现的，如代码清单 4-5 所示。

代码清单 4-5 调用 sysfs_create 函数为目录文件创建一个 inode 对象

```
int sysfs_create(struct dentry * dentry, int mode, int (*init)(struct inode *))
{
  int error = 0;
  struct inode * inode = NULL;
  if (dentry) {
      if (!dentry->d_inode) {
          struct sysfs_dirent * sd = dentry->d_fsdata;
          if ((inode = sysfs_new_inode(mode, sd))) {
               if (dentry->d_parent && dentry->d_parent->d_inode) {
                   struct inode *p_inode = dentry->d_parent->d_inode;
                   p_inode->i_mtime = p_inode->i_ctime = CURRENT_TIME;
               }
               goto Proceed;

 Proceed:
  if (init)
      error = init(inode);
  if (!error) {
      d_instantiate(dentry, inode);
      if (S_ISDIR(mode))
          dget(dentry);  /* pin only directory dentry in core */
  } else
      iput(inode);
 Done:
  return error;
}
```

sysfs_create 函数可以分成两部分：

- 第一部分是调用 sysfs_new_inode 创建一个 inode 对象。
- 第二部分调用传递进来的函数指针 init 执行初始化。

（1）调用 sysfs_new_inode 创建一个 inode 对象

先从 sysfs_new_inode 函数开始分析，它的代码如代码清单 4-6 所示。

代码清单 4-6 sysfs_new_inode 函数

```
     struct inode * sysfs_new_inode(mode_t mode, struct sysfs_dirent * sd)
124  {
125       struct inode * inode = new_inode(sysfs_sb);
126       if (inode) {
127       inode->i_blksize = PAGE_CACHE_SIZE;
128       inode->i_blocks = 0;
          /* 赋值 i_mapping 函数指针，后备设备操作信息，和 inode 函数指针 */
129       inode->i_mapping->a_ops = &sysfs_aops;
130       inode->i_mapping->backing_dev_info = &sysfs_backing_dev_info;
131       inode->i_op = &sysfs_inode_operations;
132       lockdep_set_class(&inode->i_mutex, &sysfs_inode_imutex_key);
133       /* 设置 inode 信息，主要是时间和用户信息 */
```

```
134       if (sd->s_iattr) {
135           /* sysfs_dirent has non-default attributes
136            * get them for the new inode from persistent copy
137            * in sysfs_dirent
138            */
139           set_inode_attr(inode, sd->s_iattr);
140       } else
141           set_default_inode_attr(inode, mode);
142     }
143     return inode;
144 }
```

sysfs_new_inode 函数本身很简单，但是涉及很多函数指针，这些函数指针牵扯的层面就繁杂了。backing_dev_info 和块设备的预读算法和队列控制有关，暂不在此处讨论。代码第 129 行的 sysfs_aops 和文件的 page cache 有关，是文件 page cache 的读写执行函数，在 sysfs 文件系统里，实际上并没有使用。

（2）调用 init_dir 函数执行初始化

代码第 131 行赋值了 sysfs_inode_operations，作为 inode 结构的操作函数，但是随后 sysfs_create 马上调用函数指针 init 指向的函数。这个函数就是 init_dir，它的作用是重新初始化。init_dir 的代码如代码清单 4-7 所示。

代码清单 4-7　调用 init_dir 函数执行初始化

```
static int init_dir(struct inode * inode)
{
  inode->i_op = &sysfs_dir_inode_operations;
  inode->i_fop = &sysfs_dir_operations;

  /* directory inodes start off with i_nlink == 2 (for "." entry) */
  inode->i_nlink++;
  return 0;
}
```

函数 init_dir 重新执行了赋值，它为 inode 结构赋值的函数指针组是 sysfs_dir_inode_operations。函数指针组 sysfs_dir_inode_operations 包含了一个 lookup 函数。

此时可以回顾文件的打开过程，需要文件系统提供的 lookup 函数去真正查找文件。对 sysfs 文件系统而言，lookup 函数就是 sysfs_dir_inode_operations 包含的 sysfs_lookup 函数。本章后面将继续分析 sysfs 文件系统打开文件的过程。

4.1.3　普通文件的创建

除了目录文件之外，sysfs 文件系统还定义了几种文件，分别是：普通文件、二进制文件和符号链接文件。这些文件中，普通文件最具有典型性，因此选择普通文件进行分析。

普通文件通过 sysfs_create_file 函数创建，它的代码如代码清单 4-8 所示。

代码清单 4-8 通过 sysfs_create_file 函数创建普通文件

```
int sysfs_create_file(struct kobject * kobj, const struct attribute * attr)
{
  BUG_ON(!kobj || !kobj->dentry || !attr);
  /* 指定是一个 SYSFS_KOBJ_ATTR 类型，这里创建的文件是为 sysfs 文件系统服务的 */
  return sysfs_add_file(kobj->dentry, attr, SYSFS_KOBJ_ATTR);

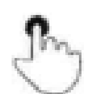
}
Int sysfs_add_file(struct dentry * dir, const struct attribute * attr, int type)
{
  struct sysfs_dirent * parent_sd = dir->d_fsdata;
  umode_t mode = (attr->mode & S_IALLUGO) | S_IFREG;
  int error = -EEXIST;

  mutex_lock(&dir->d_inode->i_mutex);
  if (!sysfs_dirent_exist(parent_sd, attr->name))
      error = sysfs_make_dirent(parent_sd, NULL, (void *)attr,
                     mode, type);
  mutex_unlock(&dir->d_inode->i_mutex);

  return error;
}
```

普通文件的创建很简单，和目录文件的创建一样，调用 sysfs_make_dirent 函数创建一个 sysfs_dirent 结构。

读者是否发现和创建目录的不同？创建目录的时候要调用 lookup_one_len 函数，lookup_one_len 函数要为新目录创建 dentry，而创建文件并没有调用 lookup_one_len 函数，也就是说创建文件的时候没创建它的 dentry 对象。



注意

这里有必要串联一下文件系统的知识点。从最简单文件系统 aufs 的例子我们知道，文件系统为每个文件（目录也是一种文件）创建了一个 inode 对象和 dentry 对象（也有特殊的文件系统例外）。而 sysfs 文件系统在创建文件的时候只创建了 sysfs_dirent 对象，那么在何时创建 dentry 和 inode？实际是在打开文件的过程中创建的。

4.2 sysfs 文件的打开操作

回顾一下，前面已经分析过 VFS 虚拟文件系统文件打开的过程。打开文件的过程，实际就是将文件的整个路径名层层解析，最终找到目标文件的过程。如果文件曾经被打开过，dentry cache 中可能保存文件的 dentry 结构（dentry cache 也有可能为节省内存释放保存的结构，此处假定没有释放）；如果在 dentry cache 中不能找到文件的 dentry 结构，那么要调用 real_lookup 函数，real_lookup 函数里再调用文件系统提供的 lookup 函数，所以在分析 sysfs

文件打开操作之前，有必要分析一下 real_lookup 调用。

4.2.1 real_lookup 函数详解

real_lookup 函数在第 2 章并没有分析，本节从 real_lookup 函数开始分析 sysfs 文件系统的文件打开操作，它的代码如代码清单 4-9 所示。

代码清单 4-9 real_lookup 函数

```
static struct dentry * real_lookup(struct dentry * parent,
struct qstr * name, struct nameidata *nd)
{
  struct dentry * result;
  struct inode *dir = parent->d_inode;

  mutex_lock(&dir->i_mutex);
  /* 再执行一次 d_lookup，检查是否有另外的进程创建文件 */
  result = d_lookup(parent, name);
  if (!result) {
      struct dentry * dentry = d_alloc(parent, name);
      result = ERR_PTR(-ENOMEM);
      if (dentry) {
          result = dir->i_op->lookup(dir, dentry, nd);
          if (result)
              dput(dentry);
          else
              result = dentry;
      }
      mutex_unlock(&dir->i_mutex);
      ……/* 此处省略校验的代码 */
      return result;
}
```

根据代码中的解释，real_lookup 需要在 dentry cache 中再搜索一遍，这是为了防止在等信号量的时候，已经有其他用户创建了文件。

对于搜索不到的文件，real_lookup 创建了一个 dentry 对象，然后调用文件系统提供的 lookup 函数执行搜索。对 sysfs 文件系统来说，就是 sysfs_lookup 函数。

4.2.2 为文件创建 inode 结构

sysfs 创建文件的时候并没有创建 dentry 对象，那么在此处为文件创建了 dentry，还需要为文件创建 inode 结构，这是 sysfs_lookup 函数的功能，它的代码如代码清单 4-10 所示。

代码清单 4-10 sysfs_lookup 函数为文件创建 inode 结构

```
static struct dentry * sysfs_lookup(struct inode *dir,
               struct dentry *dentry,struct nameidata *nd)
{
  struct sysfs_dirent * parent_sd = dentry->d_parent->d_fsdata;
```

```
    struct sysfs_dirent * sd;
    int err = 0;
    list_for_each_entry(sd, &parent_sd->s_children, s_sibling) {
         /*SYSFS_NOT_PINNED 代表文件、二进制文件和符号链接，如果不是这些，就是目录，直接退出 */
        if (sd->s_type & SYSFS_NOT_PINNED) {
            const unsigned char * name = sysfs_get_name(sd);

            if (strcmp(name, dentry->d_name.name))
                continue;
            /* 如果是符号链接文件，则调用 sysfs_attach_link*/
            if (sd->s_type & SYSFS_KOBJ_LINK)
                 err = sysfs_attach_link(sd, dentry);
            else
                 err = sysfs_attach_attr(sd, dentry);
            break;
        }
    }

    return ERR_PTR(err);
}
```

sysfs_lookup 函数首先遍历父对象 sysfs_dirent 的链表，逐一比较父对象的子成员，寻找名字和指定名字相同的子成员。这个过程和 dentry 的搜索过程类似，可见 sysfs 文件系统通过 sysfs_dirent 对象来管理文件系统的树形结构，sysfs_dirent 对象的部分功能和 dentry 的功能类似。

4.2.3 为 dentry 结构绑定属性

对于不同类型的文件，其 sysfs_dirent 对象也具有不同的属性，需要为 dentry 结构绑定各自的属性。对于普通的文件，调用 sysfs_attach_attr 函数来绑定属性，它的代码如代码清单 4-11 所示。

代码清单 4-11 调用 sysfs_attach_attr 函数绑定属性

```
static int sysfs_attach_attr(struct sysfs_dirent * sd, struct dentry * dentry)
{
  struct attribute * attr = NULL;
  struct bin_attribute * bin_attr = NULL;
  int (* init) (struct inode *) = NULL;
  int error = 0;
      /*sd 保存了文件的一些私有数据。普通文件是 attribute 结构，而二进制文件是 bin_attribute 结构 */
      if (sd->s_type & SYSFS_KOBJ_BIN_ATTR) {
            bin_attr = sd->s_element;
            attr = &bin_attr->attr;
      } else {
            attr = sd->s_element;
            init = init_file;
      }
```

```
    dentry->d_fsdata = sysfs_get(sd);
    sd->s_dentry = dentry;
    error = sysfs_create(dentry, (attr->mode & S_IALLUGO) | S_IFREG, init);
    if (error) {
        sysfs_put(sd);
        return error;
    }

        if (bin_attr) {
        dentry->d_inode->i_size = bin_attr->size;
        dentry->d_inode->i_fop = &bin_fops;
    }
    dentry->d_op = &sysfs_dentry_ops;
    /*dentry 加入 dentry cache，下次就可以在 dentry cache 中找到 */
    d_rehash(dentry);

    return 0;
}
```

sysfs_attach_attr 首先调用 sysfs_create 函数创建 inode 对象。代码执行到此，文件的 dentry 和 inode 对象都创建完成。

然后要区分二进制文件还是普通文件。如果是二进制文件，需要重新赋值文件的操作函数。在创建 inode 对象时，已经为普通文件赋值了操作函数，这些操作函数就是 sysfs 文件系统提供的 sysfs_file_operations 结构。但是二进制文件的操作函数不同，所以还需要重新赋值。

4.2.4 调用文件系统中的 open 函数

返回文件打开的过程，执行完 real_lookup 之后，还需要在 dentry_open 函数中执行文件系统提供的 open 函数。

对普通文件而言，sysfs 文件系统提供的 open 函数就是 sysfs_open_file，二进制文件的 open 函数和普通文件的基本是类似的。所以我们以普通文件的 sysfs_open_file 为准，如代码清单 4-12 所示。

代码清单 4-12 sysfs_open_file 函数

```
static int sysfs_open_file(struct inode * inode, struct file * filp)
{
  return check_perm(inode,filp);
}
```

sysfs_open 函数是 check_perm 函数的封装函数，check_perm 函数如代码清单 4-13 所示。

代码清单 4-13 check_perm 函数

```
static int check_perm(struct inode * inode, struct file * file)
{
```

```
struct kobject *kobj = sysfs_get_kobject(file->f_dentry->d_parent);
struct attribute * attr = to_attr(file->f_dentry);
struct sysfs_buffer * buffer;
struct sysfs_ops * ops = NULL;
int error = 0;

if (!kobj || !attr)
    goto Einval;

/* Grab the module reference for this attribute if we have one */
if (!try_module_get(attr->owner)) {
    error = -ENODEV;
    goto Done;
}

/*ops 实际上提供了文件的读写函数指针，后面可以见到它的应用 */
/* if the kobject has no ktype, then we assume that it is a subsystem
 * itself, and use ops for it.
 */
if (kobj->kset && kobj->kset->ktype)
    ops = kobj->kset->ktype->sysfs_ops;
else if (kobj->ktype)
    ops = kobj->ktype->sysfs_ops;
else
    ops = &subsys_sysfs_ops;
```

check_perm 函数第一部分主要作用是检查文件的权限。

sysfs 文件系统的普通文件默认具有 attribute 结构，而目录文件默认具有一个 kobject 结构。如果目录文件的 kobject 提供了文件的操作函数，普通文件要赋值为 kobject 结构提供的函数；否则就要赋值为子系统函数指针结构 subsys_sysfs_ops。然后根据文件的读写权限设置，分别检查 inode 结构的权限模式。

```
/* No error? Great, allocate a buffer for the file, and store it
 * it in file->private_data for easy access.
 */
buffer = kzalloc(sizeof(struct sysfs_buffer), GFP_KERNEL);
if (buffer) {
     init_MUTEX(&buffer->sem);
     buffer->needs_read_fill = 1;
     buffer->ops = ops;
     file->private_data = buffer;
} else
     error = -ENOMEM;
 goto Done;
}
```

check_perm 函数第二部分创建一个私有的数据结构 buffer，文件的 private_data 指针指向了这个结构。这是一种在文件对象中保存文件系统特殊信息的方式，在实际的文件系统中，经常可以看到这种使用方式。

4.3 sysfs 文件的读写

sysfs 是在内存中存在的文件系统，它的文件都只在内存中存在。因此对文件的读写实际是对内存的读写，不涉及对硬盘的操作。

4.3.1 读文件的过程分析

对文件的读仍然以普通文件为准进行分析。普通文件的读函数是 sysfs_read_file，它的代码如代码清单 4-14 所示。

代码清单 4-14　读函数 sysfs_read_file

```
static ssize_t
sysfs_read_file(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
  /* 获取 buffer 对象 */
  struct sysfs_buffer * buffer = file->private_data;
  ssize_t retval = 0;

  down(&buffer->sem);
  if (buffer->needs_read_fill) {
      if ((retval = fill_read_buffer(file->f_dentry,buffer)))
          goto out;
  }
  ....../* 省略无关的输出代码 */
  retval = flush_read_buffer(buffer,buf,count,ppos);
```

sysfs_read_file 函数调用 fill_read_buffer 申请内存页，然后填充数据，最后将数据从 buffer 复制到用户的缓存。

文件的内容是通过 fill_read_buffer 填入的，它的代码如代码清单 4-15 所示。

代码清单 4-15　调用 fill_read_buffer 函数申请内存页

```
static int fill_read_buffer(struct dentry * dentry, struct sysfs_buffer * buffer)
{
  /* 获取 sysfs_dirent 结构和属性结构，以及文件所在目录的 kobject 结构 */
  struct sysfs_dirent * sd = dentry->d_fsdata;
  struct attribute * attr = to_attr(dentry);
  struct kobject * kobj = to_kobj(dentry->d_parent);
  struct sysfs_ops * ops = buffer->ops;
  int ret = 0;
  ssize_t count;

  if (!buffer->page)
      buffer->page = (char *) get_zeroed_page(GFP_KERNEL);
  if (!buffer->page)
      return -ENOMEM;

  buffer->event = atomic_read(&sd->s_event);
```

```
  count = ops->show(kobj,attr,buffer->page);
  buffer->needs_read_fill = 0;
```

fill_read_buffer 函数首先申请一个内存页，然后调用 ops->show 向内存页填充数据。为了帮助理解，有必要从内核找个实际的例子 show 函数，既要简单又能说明问题。因此从 input 设备驱动找到一个简单的 show 函数，就是 input_dev_show_id，这个函数的作用是填充设备的 ID 名，它的代码如代码清单 4-16 所示。

代码清单 4-16 input_dev_show_id 函数填充设备的 ID 名

```
#define INPUT_DEV_ID_ATTR(name)
static ssize_t input_dev_show_id_##name(struct class_device *dev, char *buf)
{
  struct input_dev *input_dev = to_input_dev(dev);
  return scnprintf(buf, PAGE_SIZE, "%04x\n", input_dev->id.name);
}
static CLASS_DEVICE_ATTR(name, S_IRUGO, input_dev_show_id_##name, NULL);
```

字符“##”是内核中常用的一种定义方式，作用是顶替字符串。如果输入参数 name 是 bustype，函数的真正名字是 input_dev_show_id_bustype。input_dev_show_id 函数的作用是把设备 ID 的 name 复制到 buf 处，最终 sysfs_read_file 函数要把 buf 的内容复制到用户输入的用户态缓存。这样用户读到的就是 buf 里面填充的字符串。

至此，sysfs 文件的读过程分析完毕。

4.3.2 写文件的过程分析

普通文件的写函数是 sysfs_write_file，它的代码如代码清单 4-17 所示。

代码清单 4-17 写函数 sysfs_write_file

```
static ssize_t sysfs_write_file(struct file *file, const char __user *buf,
size_t count, loff_t *ppos)
{
  /* 获得 buffer 对象 */
  struct sysfs_buffer * buffer = file->private_data;
  ssize_t len;

  down(&buffer->sem);
  /* 从用户的 buf 复制数据到 buffer 对象的页 */
  len = fill_write_buffer(buffer, buf, count);
  if (len > 0)
      len = flush_write_buffer(file->f_dentry, buffer, len);
  if (len > 0)
      *ppos += len;
  up(&buffer->sem);
  return len;
}
```

sysfs_write_file 和读文件的处理类似，首先是从用户输入的 buf 复制数据到 buffer 对象的内存页，然后调用 flush_write_buffer 把用户数据填入文件的 attribute 结构。

flush_write_buffer 的代码如代码清单 4-18 所示。

代码清单 4-18 flush_write_buffer 函数

```
static int
flush_write_buffer(struct dentry * dentry, struct sysfs_buffer * buffer, size_t count)
{
  struct attribute * attr = to_attr(dentry);
  struct kobject * kobj = to_kobj(dentry->d_parent);
  struct sysfs_ops * ops = buffer->ops;

  return ops->store(kobj,attr,buffer->page,count);
}
```

函数 flush_write_buffer 最终调用了 ops 提供的 store 函数。



注意

因为读函数调用的是 show 函数，可以推理，写函数调用的 store 函数应该就是 show 函数的反过程，读者可以从内核找一个例子，自行分析一下。

4.4 kobject 结构

kobject 结构是内核定义的一种特殊结构，和 sysfs 文件系统联系很紧密。前文 sysfs 创建目录时，传递的参数就是一个 kobject 结构。实际上，可以认为 kobject 代表 sysfs 文件系统的一个目录。

4.4.1 kobject 和 kset 的关系

kset 结构里封装了一个 kobject 结构，同时包括一个链表头，属于这个 kset 的所有 kobject 都要链接到 kset 的链表头。kobject 和 kset 的关系如图 4-2 所示。

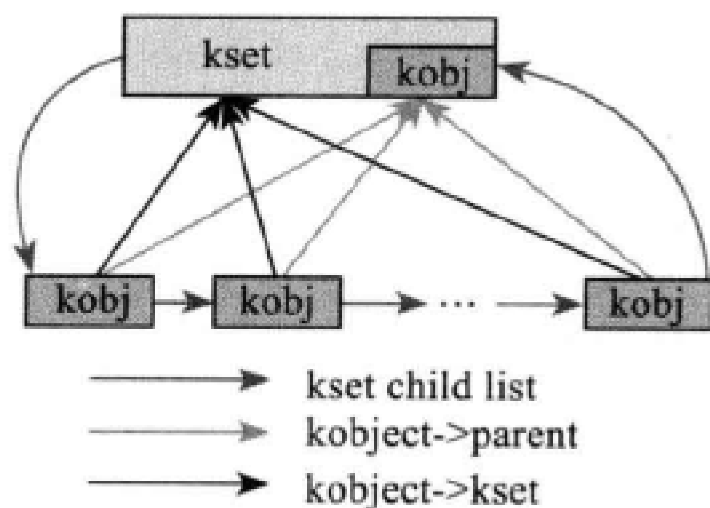


图 4-2 kset 和 kobject 关系图

4.4.2 kobject 实例：总线的注册

为了理解 kobject，我们从内核挑选一个实例，就是总线的注册。这个例子极好地解释了 kobject 的应用。第 3 章已经分析过，总线对设备和驱动具有重要作用。本章引入这个例子，既可以解释 kobject 结构的使用，又解释了总线的一些重要概念，从而为下文的分析学习打好基础。对这个例子的分析从总线的注册开始。

总线的注册使用 platform_bus_init 函数，它的代码如代码清单 4-19 所示。

代码清单 4-19 使用 platform_bus_init 函数注册总线

```
int __init platform_bus_init(void)
{
  device_register(&platform_bus);
  return bus_register(&platform_bus_type);
}
```

platform_bus_init 函数的第一部分是设备注册函数 device_register。设备是 Linux 系统比较复杂的概念，此处先跳过。

然后是总线注册函数 bus_register，它的作用是把总线对象注册到内核，代码如代码清单 4-20 所示。

代码清单 4-20 调用 bus_register 函数把总线对象注册到内核

```
int bus_register(struct bus_type * bus)
{
  int retval;
  /* 给 kobject 赋名字。这个 kobject 结构就是 bus 类型内含的 kobject 对象 */
  retval = kobject_set_name(&bus->subsys.kset.kobj, "%s", bus->name);
  if (retval)
      goto out;
  /* 子系统注册。注册过程是往 sysfs 文件系统写入一个目录 */
  subsys_set_kset(bus, bus_subsys);
  retval = subsystem_register(&bus->subsys);
  if (retval)
      goto out;
  /*devices 是一个 kset 对象，给 devices 内含的 kobject 对象赋名字，然后注册 */
  kobject_set_name(&bus->devices.kobj, "devices");
  bus->devices.subsys = &bus->subsys;
  retval = kset_register(&bus->devices);
  if (retval)
      goto bus_devices_fail;
  /*drivers 注册和 devices 注册类似 */
  kobject_set_name(&bus->drivers.kobj, "drivers");
  bus->drivers.subsys = &bus->subsys;
  bus->drivers.ktype = &ktype_driver;
  retval = kset_register(&bus->drivers);
  if (retval)
      goto bus_drivers_fail;
   ....../* 省略 klist 初始化代码 */
  bus_add_attrs(bus);
```

bus 对象内含两个 kset，一个是 devices，另一个是 drivers。devices 代表总线包含的设备对象，而 drivers 代表总线包含的驱动对象。bus 对象自身是一个 subsystem 结构，这个结构和 kset 基本是一回事，只是多了一个信号量成员而已。

bus_register 函数可以总结为三个注册。

- 第一是注册 bus 对象自身；
- 第二是注册 bus 内的设备对象；
- 第三是注册 bus 内的驱动对象。

注册 bus 对象自身使用了 subsystem_register 函数，而设备和驱动的注册都调用了 kset_register 函数。因为 subsystem 结构和 kset 基本相同，它们的注册函数也相似。所以本文选择 subsystem_register 为例进行分析，它的代码如代码清单 4-21 所示。

代码清单 4-21 使用 subsystem_register 函数注册 bus 对象自身

```
int subsystem_register(struct subsystem * s)
{
  int error;

  subsystem_init(s);
  if (!(error = kset_add(&s->kset))) {
```

subsystem_register 函数进行初始化之后，调用 kset_add 完成 kset 结构的注册功能。函数 kset_addr 的功能是在 sysfs 文件系统增加一个目录文件，它的代码如代码清单 4-22 所示。

代码清单 4-22 调用 kset_add 函数在 sysfs 文件系统增加一个目录文件

```
int kset_add(struct kset * k)
{   /* 如果没父类，使用 subsys 的 kobj*/
  if (!k->kobj.parent && !k->kobj.kset && k->subsys)
      k->kobj.parent = &k->subsys->kset.kobj;

  return kobject_add(&k->kobj);
}
```

kset_add 函数封装了 kobject_add 函数，kobject_add 执行增加一个目录文件的功能，它的代码如代码清单 4-23 所示。

代码清单 4-23 kobject_add 函数

```
int kobject_add(struct kobject * kobj)
{
  int error = 0;
  struct kobject * parent;
  /* 获取 kobj 的父结构 */
  parent = kobject_get(kobj->parent);
  ....../* 省略参数检查的代码 */

  if (kobj->kset) {
```

```
        spin_lock(&kobj->kset->list_lock);

        if (!parent)
            parent = kobject_get(&kobj->kset->kobj);
        /* kobj 链接到 kset 链表的尾部 */
        list_add_tail(&kobj->entry,&kobj->kset->list);
        spin_unlock(&kobj->kset->list_lock);
    }
    kobj->parent = parent;

    /* 创建一个目录 */
    error = create_dir(kobj);
}
```

kobject_add 函数调用 create_dir 函数创建一个目录文件。create_dir 函数的输入参数是 kobj 结构，要为这个结构创建一个目录文件，它的代码如代码清单 4-24 所示。

代码清单 4-24　调用 create_dir 函数创建一个目录文件

```
static int create_dir(struct kobject * kobj)
{
  int error = 0;
  if (kobject_name(kobj)) {
      error = sysfs_create_dir(kobj);
      if (!error) {
          /* 创建隶属于 kobj 的属性文件 */
          if ((error = populate_dir(kobj)))
```

create_dir 函数很简单，它调用了 sysfs 文件系统的 sysfs_create_dir 函数创建了一个目录文件。sysfs_create_dir 函数在本章前面已分析过。

populate_dir 函数的作用是根据 kobj 结构内含有的 attribute 结构创建文件，对每个 attribute 结构都调用 sysfs 文件系统创建文件的 sysfs_create_file 函数创建一个文件。

返回到 subsystem_register 函数，它的最终作用就是创建一个目录，目录名就是总线的名字，在这个目录下又创建一些属性文件。而 kset_register 函数的作用也是创建一个目录。这样就清楚了，bus_register 实际创建了一个名字和总线名相同的目录，在这个目录下又创建了 devices 和 drivers 两个目录。这些目录和目录下的属性文件，共同展示了一条总线和总线的设备以及驱动的属性信息。

4.5　本章小结

内核中类似 sysfs 文件系统还有不少，比如 ramfs、debugfs、configfs、proc 等。利用本章所学的知识点，读者可以自行分析这些文件系统，了解它们文件和目录的组织方式以及文件的读写方式。

第 5 章

字符设备和 input 设备

Linux 操作系统把设备划分为字符设备和块设备（Linux 操作系统的网络设备是特殊的，既不是字符设备，也不是块设备，而是一个单独的类型）。对从事 Linux 驱动的程序员来说，要么是字符设备驱动，要么是块设备驱动。

一个字符设备可以非常简单，以至于很多程序员把字符设备当作系统控制的一种手段，通过字符设备的 I/O control 函数与内核交换数据。但是实际上，Linux 内核系统很多时候是把字符设备当作一个框架来用，这种用法就复杂多了。

那么什么是字符设备和块设备？本章我们分析字符设备。

5.1 文件如何变成设备

回顾第 2 章介绍的最简单文件系统 aufs，通过 aufs_get_inode 为每个文件创建它的 inode 对象。可以看到，对文件和目录都有各自的文件操作函数和 inode 操作函数。但是默认情况下，我们用一个 init_special_inode 函数给对象赋值。

5.1.1 init_special_inode 函数

通过 init_special_inode 函数使文件变成设备，字符设备和块设备开始浮出海面，所以有必要首先分析这个函数，它的代码如代码清单 5-1 所示。

代码清单 5-1 init_special_inode 函数

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
  inode->i_mode = mode;
  if (S_ISCHR(mode)) {
      inode->i_fop = &def_chr_fops;
      inode->i_rdev = rdev;
  } else if (S_ISBLK(mode)) {
      inode->i_fop = &def_blk_fops;
```

```
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode))
        inode->i_fop = &def_fifo_fops;
    else if (S_ISSOCK(mode))
        inode->i_fop = &bad_sock_fops;
    else
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o)\n",
              mode);
}
```

这段代码很简单，如果是字符设备，它的文件操作结构指针被赋值为 def_chr_fops；如果是块设备，则被赋值为 def_blk_fops。同时 inode 的 i_rdev 被赋值为 rdev，这个 rdev 其实就是由主设备号和从设备号生成的设备号。

通过这个特别的函数，inode 的文件函数指针 i_fop 被替换，从此 indoe 不再是普通的文件 inode，而是分别可以代表字符设备、块设备、fifo 和 socket 的特殊 inode。

5.1.2 def_chr_fops 结构

本章重点分析为字符设备提供的 def_chr_fops 结构，它的代码如代码清单 5-2 所示。

代码清单 5-2 def_chr_fops 结构

```
const struct file_operations def_chr_fops = {
  .open = chrdev_open,
};
int chrdev_open(struct inode * inode, struct file * filp)
{
  struct cdev *p;
  struct cdev *new = NULL;
  int ret = 0;

  spin_lock(&cdev_lock);
  p = inode->i_cdev;

  /* 如果字符设备不存在 */
  if (!p) {
      struct kobject *kobj;
      int idx;
      spin_unlock(&cdev_lock);
      /* 通过 kobj_lookup 查找字符设备的 kobj 结构 */
      kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
      if (!kobj)
          return -ENXIO;
      /* 调用 container 方法，获得 cdev 对象 */
      new = container_of(kobj, struct cdev, kobj);
      spin_lock(&cdev_lock);
      /* 再次检查 p*/
      p = inode->i_cdev;
      if (!p) {
```

```
            /* 赋值。inode 的字符设备指针指向发现的字符设备 */
            inode->i_cdev = p = new;
            inode->i_cindex = idx;
            list_add(&inode->i_devices, &p->list);
            new = NULL;
        } else if (!cdev_get(p))
            ret = -ENXIO;
    } else if (!cdev_get(p))
        ret = -ENXIO;
    spin_unlock(&cdev_lock);
    cdev_put(new);
    if (ret)
        return ret;
   /* 获得设备的函数指针，对 input 设备来说，就是 input_fops*/
    filp->f_op = fops_get(p->ops);
    if (!filp->f_op) {
        cdev_put(p);
        return -ENXIO;
    }
   /* 这个 open 函数就是 input 设备的 input_open_file*/
    if (filp->f_op->open) {
        /* 大内核锁 */
        lock_kernel();
        ret = filp->f_op->open(inode,filp);
        unlock_kernel();
    }
    if (ret)
        cdev_put(p);
    return ret;
}
```

chrdev_open 函数前面说明了，每次打开一个字符设备的时候，都调用这个函数。它首先根据设备号调用 kobj_lookup 搜索注册的字符设备对象，如果找到字符设备，执行字符设备的 open 函数，找不到则返回错误。

Linux 系统提供了 mknod 程序，使用这个程序用户可以根据主从设备号创建特殊文件，比如字符设备文件或者块设备文件。从内核的角度分析，mknod 为特殊文件创建了一个 inode 结构和 dentry 结构，inode 结构的成员包含主从设备号和设备类型，然后调用 init_special_inode 函数为设备文件的 inode 设置不同的函数指针。打开设备文件时，通过 chrdev_open 函数真正调用设备驱动本身的 open 函数，当然，前提是已经注册了设备驱动函数。

5.2 input 设备的注册

为了正确理解和应用，需要一个典型的字符设备示例。本节直接从内核选择一个例子，就是 input 设备。这个设备不但典型，而且具有很大实用价值。input 是一个虚拟的设备，在 Linux 系统中，键盘、鼠标、触摸屏和游戏杆都要由 input 设备统一管理。

input 设备是个字符设备，它是如何注册设备驱动的？这要从 input 设备的初始化函数 input_init 开始。

5.2.1 主从设备号

Linux 系统通过设备号来区分不同的设备。设备号由两部分组成：主设备号和从设备号。下面摘录了系统定义的一些主设备号（来自 include/linux/major.h）。

```
#define UNNAMED_MAJOR          0
#define MEM_MAJOR              1
#define RAMDISK_MAJOR          1
#define FLOPPY_MAJOR           2
#define PTY_MASTER_MAJOR       2
#define IDE0_MAJOR             3
#define HD_MAJOR               IDE0_MAJOR
#define PTY_SLAVE_MAJOR        3
#define TTY_MAJOR              4
#define TTYAUX_MAJOR           5
#define LP_MAJOR               6
#define VCS_MAJOR              7
#define LOOP_MAJOR             7
#define SCSI_DISK0_MAJOR       8
#define SCSI_TAPE_MAJOR        9
#define MD_MAJOR               9
#define MISC_MAJOR             10
#define SCSI_CDROM_MAJOR       11
#define MUX_MAJOR              11
#define XT_DISK_MAJOR          13
#define INPUT_MAJOR            13
```

系统定义了多个主设备号，本节要讨论的 input 设备占第 13 号主设备号。从设备号区分归属于同一个主设备的独立设备。比如，系统中有几个硬盘，它们占用了不同的次设备号。

也许读者会想，input 设备包括各种各样的输入设备。别说键盘、鼠标的不同，就是鼠标之间也有各种各样的型号，难道这些键盘、鼠标、游戏杆都用的同一个驱动？

这个问题，从内核的角度很容易理解。实际上，字符设备 input 是设备的一个聚合层，众多的驱动和设备被 input 封装，经过这个封装层之后，键盘和鼠标等设备就各行其是，分别由不同的驱动所控制。而且不仅仅 input 是一个封装层，在 input 之下，系统还提供了几个层次的封装。这是 Linux 内核设计的一个重要思想。在内核代码的阅读过程中，我们将发现越来越多的这种分层架构。这也是 Linux 内核设计向面向对象设计靠拢的一个标志，在本文的叙述中，也大量使用“对象”这个词来描述内核中各个层次生成的数据结构。以一个普通的键盘来说，因为键盘属于串口设备（假定，当然也有 USB 键盘等），所以内核要创建一个串口设备，同时安装相应的串口驱动。而键盘属于 input 设备范畴，它也要创建一个 input 设备，并安装相应的 input 驱动。

5.2.2 把 input 设备注册到系统

input_init 函数的作用是把 input 设备注册到系统，它的代码如代码清单 5-3 所示。

代码清单 5-3 input_init 函数

```
static int __init input_init(void)
{
  int err;
  /*input 要注册 input 类，这部分先跳过 */
  err = class_register(&input_class);
  if (err) {
      printk(KERN_ERR "input: unable to register input_dev class\n");
      return err;
  }
  /* 在 proc 目录下创建 input 相关的文件 */
  err = input_proc_init();
  if (err)
      goto fail1;

  err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
```

input_init 函数最终调用 register_chrdev 函数来注册 input 驱动，它的代码如代码清单 5-4 所示。

代码清单 5-4 register_chrdev 函数

```
int register_chrdev(unsigned int major, const char *name,
          const struct file_operations *fops)
{
  struct char_device_struct *cd;
  struct cdev *cdev;
  char *s;
  int err = -ENOMEM;

  cd = __register_chrdev_region(major, 0, 256, name);
  if (IS_ERR(cd))
      return PTR_ERR(cd);

  /* 申请一个 cdev 对象 */
  cdev = cdev_alloc();
  if (!cdev)
      goto out2;

  cdev->owner = fops->owner;
  cdev->ops = fops;
  /* 设置字符设备 kobj 结构的名字 */
  kobject_set_name(&cdev->kobj, "%s", name);
  for (s = strchr(kobject_name(&cdev->kobj),'/'); s; s = strchr(s, '/'))*s = '!';

  /*cdev 插入链表 */
  err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
```

register_chrdev 函数实际执行了两个登记，一个是登记设备的区间，另一个登记是注册一个字符设备。首先分析设备区间的登记。

5.2.3 设备区间的登记

区间是主设备号和从设备号共同占有的一段空间，register_chrdev 函数要登记 0 ~ 256 的从设备号区间，这个区间之前不能被占用。登记区间通过 __register_chrdev_region 函数实现，它的代码如代码清单 5-5 所示。

代码清单 5-5 __register_chrdev_region 函数

```
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
             int minorct, const char *name)
{
  struct char_device_struct *cd, **cp;
  int ret = 0;
  int i;

  cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);
  if (cd == NULL)
      return ERR_PTR(-ENOMEM);

  mutex_lock(&chrdevs_lock);

/* 主设备号为 0，说明这个设备没指定设备号，需要分配一个 */
  if (major == 0) {
      for (i = ARRAY_SIZE(chrdevs)-1; i > 0; i--) {
          if (chrdevs[i] == NULL)
              break;
      }

      if (i == 0) {
          ret = -EBUSY;
          goto out;
      }
      major = i;
      ret = major;
  }
```

1）__register_chrdev_region 函数第一部分首先创建一个 char_device_struct 结构，然后要考虑输入的主设备号为 0 的情况。这种情况下，要为字符设备分配一个主设备号。

分配主设备号的算法是从高到低遍历数组 chrdevs，如果发现某个主设备号为空，则分配给字符设备。chrdevs 是全局变量，它是个 255 个元素的指针数组，对应设备的主设备号。如果输入的主设备号大于 255，则取其余数。这个结构数组保存了所有的主设备号和从设备号。

```
cd->major = major;
cd->baseminor = baseminor;
```

```
    cd->minorct = minorct;
    strncpy(cd->name,name, 64);

    /* 根据主设备号计算索引，实际是主设备号除以 255 的余数 */
    i = major_to_index(major);

    /* 找一个未占用的区间 */
    for (cp = &chrdevs[i]; *cp; cp = &(*cp)->next)
        if ((*cp)->major > major ||
            ((*cp)->major == major && (*cp)->baseminor >= baseminor))
             break;
    if (*cp && (*cp)->major == major &&
        (*cp)->baseminor < baseminor + minorct) {
        ret = -EBUSY;
        goto out;
    }
    cd->next = *cp;
    *cp = cd;
    mutex_unlock(&chrdevs_lock);
    return cd;
```

2）__register_chrdev_region 第二部分从数组 chrdevs 找到未占用的区间。首先通过主设备号索引获得结构 char_device_struct，然后遍历 char_device_struct 结构的单向链表，依次比较从设备号，找到一个合适的区间。最后将创建的字符设备结构 cd 链接到单向链表，完成字符设备区间的登记。

5.2.4 注册字符设备

返回 register_chrdev 函数，cdev_add 函数的功能是注册字符设备，它的代码如代码清单 5-6 所示。

代码清单 5-6 cdev_add 函数

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
  p->dev = dev;
  p->count = count;
  return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

cdev_add 函数要把复合设备号（由主设备号和从设备号计算而来）和设备区间注册到系统，这是通过调用 kobj_map 函数实现的。

kobj_map 和前一节学习的 kobj_lookup 是同一组函数，目的就是通过系统的指针数组和链表管理字符设备，kobj_map 函数的代码如代码清单 5-7 所示。

代码清单 5-7 kobj_map 函数

```
int kobj_map(struct kobj_map *domain, dev_t dev, unsigned long range,
       struct module *module, kobj_probe_t *probe,
```

```
        int (*lock)(dev_t, void *), void *data)
{
    /* 计算设备输入的 range 可能占用几个主设备号。对 256 这个 range 来说，只占用一个主设备号 */
    unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1;
    unsigned index = MAJOR(dev);
    unsigned i;
    struct probe *p;

    if (n > 255)
        n = 255;

    p = kmalloc(sizeof(struct probe) * n, GFP_KERNEL);

    if (p == NULL)
        return -ENOMEM;
    /* 为 p 赋值 */
    for (i = 0; i < n; i++, p++) {
        p->owner = module;
        p->get = probe;
        p->lock = lock;
        p->dev = dev;
        p->range = range;
        p->data = data;
    }
    mutex_lock(domain->lock);
    for (i = 0, p -= n; i < n; i++, p++, index++) {
        struct probe **s = &domain->probes[index % 255];
        while (*s && (*s)->range < range)
            s = &(*s)->next;
        p->next = *s;
        *s = p;
    }
    mutex_unlock(domain->lock);
    return 0;
}
```

系统提供了一个 kobj_map 结构的指针 cdev_map，它包含了一个指针数组 probes，数组元素是 255 个，每个主设备号对应 probes 数组的一个元素。每个 probes 结构包含一个单向链表，所有具有相同主设备号的字符设备（主设备号大于 255 要取其余数）都链接到单向链表。kobj_map 函数遍历单向链表，找到一个合适的位置，安放创建的 probe 结构，完成字符设备的注册过程。

5.2.5 打开 input 设备

对设备进行读写前，首先要打开设备。根据第 2 章文件打开过程的分析和本章字符设备的分析，在内核中打开设备最后调用了设备驱动的 open 函数。上一节分析的 input_init 函数注册了 input 设备的 open 函数，也就是 input_open_file 函数，它的代码如代码清单 5-8 所示。

代码清单 5-8 input_open_file 函数

```
static int input_open_file(struct inode *inode, struct file *file)
{
  /*根据次设备号获得 input_handler*/
  struct input_handler *handler = input_table[iminor(inode) >> 5];
  const struct file_operations *old_fops, *new_fops = NULL;
  int err;
  /* 删掉异常处理代码 */
  if (!new_fops->open) {
      fops_put(new_fops);
      return -ENODEV;
  }
  /* 函数指针被替换 */
  old_fops = file->f_op;
  file->f_op = new_fops;
  /* 调用新的 open 函数 */
  err = new_fops->open(inode, file);
```

input_open_file 函数最重要的部分是 input_handler 的应用。input_table 是个数组，包含 8 个 input_handler 指针。input 设备封装了 8 个不同的 handler，每个对应一个次设备号。设备打开时通过次设备号获得注册的 input_handler，然后调用 input_handler 提供的 open 函数。

5.3 input 设备架构

input 本身是个字符设备，但是经过层层分析，原来 input 设备里面隐藏了众多的设备和驱动，input 设备其实是设备和驱动的封装。那么这些设备和驱动是怎么注册进去的？内核为何专门做一个封装层来汇聚这些设备？内核源码根目录的 drivers/input/input.c 本身并不复杂，我们重点分析两个函数 input_register_handler 和 input_register_device 就可以了解这些问题。

5.3.1 注册 input 设备的驱动

第一步分析 input_register_handler 函数。这个函数的作用是注册 input 设备的驱动，它的代码如代码清单 5-9 所示。

代码清单 5-9 input_register_handler 函数

```
void input_register_handler(struct input_handler *handler)
{
  struct input_dev *dev;
  struct input_handle *handle;
  struct input_device_id *id;

  if (!handler)
      return;

  INIT_LIST_HEAD(&handler->h_list);
```

```
    /* 注册到 input_table, input_table 是个 8 元素的 input_handler 指针数组 */
    if (handler->fops != NULL)
        input_table[handler->minor >> 5] = handler;

    /*handler 链接到 input_handler_list 的链表头 */
    list_add_tail(&handler->node, &input_handler_list);

    list_for_each_entry(dev, &input_dev_list, node)
        if (!handler->blacklist || !input_match_device(handler->blacklist, dev))
            if ((id = input_match_device(handler->id_table, dev)))
                if ((handle = handler->connect(handler, dev, id))) {
                    input_link_handle(handle);
                    if (handler->start)
                        handler->start(handle);
                }

    input_wakeup_procfs_readers();
}
```

input_register_handler 函数代码的最后一段是遍历所有注册的 input 设备，检查能否和注册的 handler 匹配。

检查原则如下：

- 检查设备是否在 handler 的黑名单里；
- 检查 handler 的 ID 表是否和设备的 ID 表相等。

如果 handler 和设备匹配，调用 handler 提供的 connect 函数和设备建立连接。

到此可以总结，input 设备维护了两个链表，一个是设备链表，另一个是 handler 链表（handler 可以看做驱动），注册一个驱动要和所有的设备一一匹配，看是否适合。这就是 input 框架的管理机制。

5.3.2 匹配 input 管理的设备和驱动

input 管理的设备和驱动是如何匹配的？这是 input_match_device 函数实现的功能，所以有必要了解它的代码，理清匹配的依据，如代码清单 5-10 所示。

代码清单 5-10 input_match_device 函数

```
static struct input_device_id *input_match_device(struct input_device_id *id,
struct input_dev *dev)
{
  int i;

  for (; id->flags || id->driver_info; id++) {
      if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
          if (id->bustype != dev->id.bustype)
              continue;

      if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
```

```
            if (id->vendor != dev->id.vendor)
                continue;

        if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
            if (id->product != dev->id.product)
                continue;

        if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
            if (id->version != dev->id.version)
                continue;
        /* 逐一检查事件类型是否匹配 */
        MATCH_BIT(evbit,  EV_MAX);
        MATCH_BIT(keybit, KEY_MAX);
        MATCH_BIT(relbit, REL_MAX);
        MATCH_BIT(absbit, ABS_MAX);
        MATCH_BIT(mscbit, MSC_MAX);
        MATCH_BIT(ledbit, LED_MAX);
        MATCH_BIT(sndbit, SND_MAX);
        MATCH_BIT(ffbit,  FF_MAX);
        MATCH_BIT(swbit,  SW_MAX);

        return id;
    }
    return NULL;
}
```

input_match_devicenput 函数逐一对比驱动的 ID 表和设备的 ID 表，检查它们的总线类型、制造商、产品号和版本，以及事件类型是否相等。

5.3.3 注册 input 设备

分析完 input_register_handler，可以返回 input_register_device 函数。这个函数作用是注册 input 设备，它的代码如代码清单 5-11 所示。

代码清单 5-11 input_register_device 函数

```
int input_register_device(struct input_dev *dev)
{
  static atomic_t input_no = ATOMIC_INIT(0);
  struct input_handle *handle;
  struct input_handler *handler;
  struct input_device_id *id;
  const char *path;
  int error;

  set_bit(EV_SYN, dev->evbit);

/* 初始化设备的 timer*/
  init_timer(&dev->timer);
```

```
    if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
        dev->timer.data = (long) dev;
        dev->timer.function = input_repeat_key;
        dev->rep[REP_DELAY] = 250;
        dev->rep[REP_PERIOD] = 33;
    }

    /* 设备加入 input 的设备列表 */
    INIT_LIST_HEAD(&dev->h_list);
    list_add_tail(&dev->node, &input_dev_list);
```

input_register_device 函数第一部分是初始化设备，将设备加入总的 input 设备链表。这样，通过链表就可以遍历所有的 input 设备。同时初始化设备的 timer。这个 timer 的作用是设置的定时时间到达，自动重复输入 input 设备的按键值。

```
/* 指定设备属于 input 类 */
dev->cdev.class = &input_class;
snprintf(dev->cdev.class_id, sizeof(dev->cdev.class_id),
     "input%ld", (unsigned long) atomic_inc_return(&input_no) - 1);

error = class_device_add(&dev->cdev);
if (error)
    return error;

error = sysfs_create_group(&dev->cdev.kobj, &input_dev_attr_group);
if (error)
    goto fail1;

error = sysfs_create_group(&dev->cdev.kobj, &input_dev_id_attr_group);
if (error)
    goto fail2;

error = sysfs_create_group(&dev->cdev.kobj, &input_dev_caps_attr_group);
if (error)
    goto fail3;
```

input_register_device 函数第二部分通过 sysfs 文件系统创建设备的属性文件。文件在系统根目录 /sys/input/input*/ 里面。sysfs 文件系统第 4 章已经介绍过，此处略过。为增加感性认识，读者可以比较一下创建的文件是否和代码的设计一致。

```
/* 省略输出设备名字的部分代码 */
  list_for_each_entry(handler, &input_handler_list, node)
       if (!handler->blacklist || !input_match_device(handler->blacklist, dev))
           if ((id = input_match_device(handler->id_table, dev)))
                 if ((handle = handler->connect(handler, dev, id))) {
                      input_link_handle(handle);
                      if (handler->start)
                          handler->start(handle);
                 }
  input_wakeup_procfs_readers();
```

input_register_device 第三部分的代码和 input_register_handler 函数最后的代码很像，不过这次是遍历所有的驱动，检查是否和设备匹配。匹配的算法同样是先检查驱动的黑名单，再检查驱动和设备的 ID 表是否适合。

分析完 input 框架，引出了一个问题：内核为什么要加这样一个层次？所有的层次设计主要是为了复用代码，简化其他层次的工作量。input 汇聚的设备，不管是键盘、鼠标还是游戏杆触摸屏，它们的公共特征就是截获用户的输入，并交给操作系统处理。所以 input 提供了重要的事件处理函数 input_event，通过这个函数上报用户输入（实际上输入到终端的输入 buffer）。那么具体设备的驱动（如键盘驱动），只要调用 input_event 就可以上报用户的按键输入，节省了设备驱动的工作量。

5.4 本章小结

内核驱动中，类似 input 这样的框架还有一些。例如，根目录下的 sound/core/sound.c，定义了一个字符设备来统一管理声音设备；目录 drivers/video/fbmem.c，同样定义了一个字符设备来管理 frame buffer 对象。这样的架构还有很多，这些相似的架构分析掌握一种就可以触类旁通，大大减少学习内核的时间。读者可以试着分析这些驱动，从而加强对 Linux 驱动架构的理解和掌握。

在这里，有必要串联一下知识点，帮助我们更全面地理解 Linux 设备和驱动的架构。回顾前文我们知道，设备配置表、总线和驱动是整个内核设备架构的三大层次。设备配置表描述了设备本身物理特性（以 PCI 设备为例），包括设备的寄存器信息和内存信息；而总线，不管是物理上存在的 PCI 总线，或者其他并不真正存在的虚拟总线（比如 platform 总线），作为一个软件架构，它的作用是一个容器，把设备和驱动都容纳在其中。通过总线，可以发现设备、为设备发现驱动、配置设备信息。

通过本章的分析，我们了解到设备驱动本身是可以分为多个层次的，对一个键盘设备而言，它的驱动分为 input 层、虚拟键盘层驱动（input_handler 层）、真实键盘驱动层。第 7 章我们还要详细分析键盘的这种多层次结构。

第 6 章

platform 总线

从 input 设备和字符设备的分析中，我们初步理解了总线发现设备、管理设备、配置设备的功能。一般来说，总线都是物理存在的，但是 Linux 系统提供了一种简单的总线 platform。

platform 并不是一种物理存在的总线，而是个逻辑概念。现代的 PC 机系统通常提供了一条根总线（PCI 总线）管理设备，但是有些设备并没有挂载在 PCI 总线上（比如键盘、鼠标的控制器），所以不能由 PCI 总线管理，于是 Linux 内核虚拟了 platform 总线来统一管理这种设备。

6.1 从驱动发现设备的过程

platform 总线虽然是很简单的总线，但是一样具有总线的通用功能。通过分析这种简单的总线，可以帮助我们理解总线的概念和总线对设备和驱动的管理，方便后面对复杂总线的分析，比如说 PCI 总线。

我们选的例子是 q40kbd，在 drivers/input/serio 目录里。这是一个键盘控制器驱动，它使用了 platform 总线。

6.1.1 驱动的初始化

设备驱动一般从它的初始化函数开始分析，q40kbd 驱动的初始化函数是 q40kbd_init，它的作用是把驱动程序注册到系统，代码如代码清单 6-1 所示。

代码清单 6-1 q40kbd_init 函数

```
static int __init q40kbd_init(void)
{
  int error;

  if (!MACH_IS_Q40)
```

```
        return -EIO;
    /* 驱动作为platform总线驱动注册 */
    error = platform_driver_register(&q40kbd_driver);
    if (error)
        return error;

    /* 分配一个platform设备 */
    q40kbd_device = platform_device_alloc("q40kbd", -1);
    if (!q40kbd_device)
        goto err_unregister_driver;

    /*platform设备注册 */
    error = platform_device_add(q40kbd_device);
    if (error)
        goto err_free_device;

    return 0;
```

这段代码很简单，首先注册一个 platform 驱动，然后注册一个 platform 设备。这个过程显示了 platform 总线的用法。第 3 章介绍的 PCI 总线可以自动扫描设备，而 platform 总线是虚拟的总线，物理上并不存在，没有扫描设备的功能，所以 platform 总线需要直接注册设备。本节先从驱动的注册开始分析。

6.1.2 注册驱动

驱动注册调用的函数是 platform_driver_register，它的代码如代码清单 6-2 所示。

代码清单 6-2　platform_driver_register

```
int platform_driver_register(struct platform_driver *drv)
{
        drv->driver.bus = &platform_bus_type;
    if (drv->probe)
        drv->driver.probe = platform_drv_probe;
    if (drv->remove)
        drv->driver.remove = platform_drv_remove;
    if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;
    if (drv->suspend)
        drv->driver.suspend = platform_drv_suspend;
    if (drv->resume)
        drv->driver.resume = platform_drv_resume;
    return driver_register(&drv->driver);
}
```

platform_driver_register 函数把驱动的总线设置为 platform 总线，然后依次设置驱动的各个函数指针，最后调用 driver_register 函数注册驱动。driver_register 函数很简单，初始化之后就调用 bus_add_driver 函数。

6.1.3 为总线增加一个驱动

bus_add_driver 函数的名字显示，它的作用是为总线增加一个驱动。bus_add_driver 的代码如代码清单 6-3 所示。

代码清单 6-3 bus_add_driver 函数

```
int bus_add_driver(struct device_driver * drv)
{
  struct bus_type * bus = get_bus(drv->bus);
  int error = 0;

  if (bus) {
       pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
       /* 为 kobject 结构设置名字 */
       error = kobject_set_name(&drv->kobj, "%s", drv->name);
       if (error) {
            put_bus(bus);
            return error;
       }
       drv->kobj.kset = &bus->drivers;
       /* 为 sysfs 文件系统创建设备的相关文件 */
       if ((error = kobject_register(&drv->kobj))) {
            put_bus(bus);
            return error;
       }
       driver_attach(drv);
       /* 驱动加入总线的驱动列表 */
       klist_add_tail(&drv->knode_bus, &bus->klist_drivers);
       /* 这行和下面一行也是为了在 sysfs 创建驱动的属性文件和模块 */
       module_add_driver(drv->owner, drv);
       driver_add_attrs(bus, drv);
       add_bind_files(drv);
  }
  return error;
}
```

bus_add_driver 函数使用了 kobject_register 和 driver_add_attrs 等函数为 sysfs 文件系统创建设备驱动相关的目录和文件。第 4 章 sysfs 文件系统已经介绍过这种用法，就不再一一分析了。

6.1.4 驱动加载

真正执行驱动加载的是 driver_attach 函数，它的代码如代码清单 6-4 所示。

代码清单 6-4 driver_attach 函数

```
void driver_attach(struct device_driver * drv)
{
  bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
```

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
          void * data, int (*fn)(struct device *, void *))
{
  struct klist_iter i;
  struct device * dev;
  int error = 0;

  if (!bus)
       return -EINVAL;
  /* 初始化一个 klist，从设备 start 开始 */
  klist_iter_init_node(&bus->klist_devices, &i,
               (start ? &start->knode_bus : NULL));
  while ((dev = next_device(&i)) && !error)
       error = fn(dev, data);
  klist_iter_exit(&i);
  return error;
}
```

bus_for_each_dev 函数首先初始化一个 klist_iter 结构，目的是在双向链表中定位一个成员。klist 结构和使用在 lib 目录下的 klist.c 文件中定义，代码和功能都非常简单，读者可以自行阅读理解。

6.1.5 遍历总线上已经挂载的设备

遍历总线上已经挂载的设备，起始位置是初始化 klist_iter 结构时设置的 start 设备，只遍历这个设备之后挂载的设备。当前场景设置的 start 设备为空，所以要遍历所有 platform 总线的设备。对每个设备调用 fn 函数指针。fn 就是传入的函数指针 __driver_attach，它的代码如代码清单 6-5 所示。

代码清单 6-5 __driver_attach 函数

```
static int __driver_attach(struct device * dev, void * data)
{
  struct device_driver * drv = data;

  if (dev->parent) /* Needed for USB */
       down(&dev->parent->sem);
       down(&dev->sem);
       if (!dev->driver)
       driver_probe_device(drv, dev);
```

__driver_attach 获取设备的锁之后，调用 driver_probe_device 函数，它的代码如代码清单 6-6 所示。

代码清单 6-6 driver_probe_device 函数

```
int driver_probe_device(struct device_driver * drv, struct device * dev)
{
```

```
    int ret = 0;
    /* 先调用总线配置的match函数 */
    if (drv->bus->match && !drv->bus->match(dev, drv))
         goto Done;

    pr_debug("%s: Matched Device %s with Driver %s\n",
         drv->bus->name, dev->bus_id, drv->name);
         dev->driver = drv;
    /* 总线的match函数通过后，继续调用总线的probe函数 */
    if (dev->bus->probe) {
         ret = dev->bus->probe(dev);
         if (ret) {
              dev->driver = NULL;
              goto ProbeFailed;
         }
    } else if (drv->probe) {
         /* 如果驱动提供了probe函数，则调用驱动的probe函数 */
         ret = drv->probe(dev);
         if (ret) {
              dev->driver = NULL;
              goto ProbeFailed;
         }
    }
    /* 设备发现了驱动，通过sysfs创建一些文件。和设备做符号链接 */
    device_bind_driver(dev);
```

driver_probe_device 函数可以分为两个步骤。

- 第一步调用总线提供的 match 函数。如果检查通过，说明该设备和驱动是匹配的，设备所指向的驱动指针要赋值为当前驱动。
- 第二步是探测（probe）。首先调用总线提供的 probe 函数，如果驱动有自己的 probe 函数，还要调用驱动的 probe 函数。

probe 的目的是总线或者设备进一步的探测。比如硬盘控制器，它本身是个 pci 设备，同时又提供硬盘接入的功能。那么它驱动的 probe 函数就要扫描 scsi 总线，把所有接入的硬盘都扫描出来。

1. match 函数

platform 总线的 match 函数就是 platform_match，它的代码如代码清单 6-7 所示。

代码清单 6-7 platform_match 函数

```
static int platform_match(struct device * dev, struct device_driver * drv)
{
  struct platform_device *pdev = container_of(dev, struct platform_device, dev);
  return (strncmp(pdev->name, drv->name, BUS_ID_SIZE) == 0);
}
```

platform_match 函数很简单，就是比较驱动的名字和设备的名字是否相同，相同就可以

匹配。而注册的 q40kbd 驱动的名字是“q40kbd”，也就是说如果找到这个名字的设备，两者就匹配了。

2. probe 函数

platform 总线的 probe 函数是 platform_drv_probe，这个函数是个封装函数，它只是简单调用了驱动的 probe 函数。驱动的 probe 函数就是 q40kbd_probe，它的代码如代码清单 6-8 所示。

代码清单 6-8 q40kbd_probe 函数

```
static int __devinit q40kbd_probe(struct platform_device *dev)
{
  q40kbd_port = kzalloc(sizeof(struct serio), GFP_KERNEL);
  if (!q40kbd_port)
       return -ENOMEM;

  q40kbd_port->id.type    = SERIO_8042;
  q40kbd_port->open       = q40kbd_open;
  q40kbd_port->close      = q40kbd_close;
  q40kbd_port->dev.parent = &dev->dev;
  strlcpy(q40kbd_port->name, "Q40 Kbd Port", sizeof(q40kbd_port->name));
  strlcpy(q40kbd_port->phys, "Q40", sizeof(q40kbd_port->phys));

  serio_register_port(q40kbd_port);
  printk(KERN_INFO "serio: Q40 kbd registered\n");

  return 0;
}
```

q40kbd_probe 函数设置了一个 serio 结构变量，然后注册到系统。调用 serio_register_port 这关键的一步是实现什么？在 platform 总线里面又注册 serio，又是为什么？这些问题放到下一章分析。

6.2 从设备找到驱动的过程

总结 platform 总线驱动的注册过程，我们发现和 input 设备驱动注册的过程很相像，都是逐个遍历设备，检查是否和驱动匹配。由此可以联想，platform 总线加载设备的过程，应该也是遍历驱动，看是否和设备匹配。事实是否如此？我们从设备注册的过程开始分析。

6.2.1 注册设备和总线类型

注册设备使用的是 platform_device_add 函数，它的代码如代码清单 6-9 所示。

代码清单 6-9 platform_device_add 函数

```
int platform_device_add(struct platform_device *pdev)
{
  int i, ret = 0;

  if (!pdev)
       return -EINVAL;
  /* 如果没父设备，就设置 platform_bus 为父设备 */
  if (!pdev->dev.parent)
       pdev->dev.parent = &platform_bus;
  /* 设置设备的 bus 为 platform_bus_type */
  pdev->dev.bus = &platform_bus_type;

  if (pdev->id != -1)
       snprintf(pdev->dev.bus_id, BUS_ID_SIZE, "%s.%u", pdev->name, pdev->id);
  else
       strlcpy(pdev->dev.bus_id, pdev->name, BUS_ID_SIZE);
```

platform_device_add 函数第一部分是设置设备的父设备和总线类型。

6.2.2 注册设备的资源

platform_device_add 函数第二部分注册设备的资源。

```
/* 把设备 I/O 端口和 I/O 内存资源注册到系统 */
for (i = 0; i < pdev->num_resources; i++) {
     struct resource *p, *r = &pdev->resource[i];

     if (r->name == NULL)
          r->name = pdev->dev.bus_id;

     p = r->parent;
     if (!p) {
          if (r->flags & IORESOURCE_MEM)
               p = &iomem_resource;
          else if (r->flags & IORESOURCE_IO)
               p = &ioport_resource;
     }

     if (p && insert_resource(p, r)) {
          printk(KERN_ERR
               "%s: failed to claim resource %d\n",
               pdev->dev.bus_id, i);
          ret = -EBUSY;
          goto failed;
     }
}
```

注意 I/O 端口（设备控制寄存器）和 I/O 内存注册到系统的部分代码。回顾设备基本概念

一章，PCI 总线通过扫描设备的配置文件，可以配置设备的 I/O 端口和 I/O 内存，而 platform 总线在物理上并不存在，自然不能自动扫描设备，那么如何配置设备的 I/O 端口和 I/O 内存？

从内核驱动中，可以发现 platform 设备很多使用了探测的方式。一般是先往一个 I/O 端口写数据，看是否回应判断设备的 I/O 端口是否存在。至于本章分析的 q40kbd 设备，它甚至没有注册自己的 I/O 端口和内存，而是直接使用了缺省值。说明这是一种很古老的设备了。

```
pr_debug("Registering platform device '%s'. Parent at %s\n",
     pdev->dev.bus_id, pdev->dev.parent->bus_id);
ret = device_add(&pdev->dev);
```

6.2.3 增加一个设备对象

platform_device_add 函数第三部分调用 device_add 增加一个设备对象，它的代码如代码清单 6-10 所示。

代码清单 6-10 device_add 函数

```
int device_add(struct device *dev)
{
  struct device *parent = NULL;
  char *class_name = NULL;
  int error = -EINVAL;

  dev = get_device(dev);
  if (!dev || !strlen(dev->bus_id))
       goto Error;
  /* 获得父设备 */
  parent = get_device(dev->parent);

  pr_debug("DEV: registering device: ID = '%s'\n", dev->bus_id);

  /* first, register with generic layer. */
  kobject_set_name(&dev->kobj, "%s", dev->bus_id);
  if (parent)
       dev->kobj.parent = &parent->kobj;
  /* 在 sys 目录生成设备目录 */
  if ((error = kobject_add(&dev->kobj)))
       goto Error;
  /* 设置 uevent 属性 */
  dev->uevent_attr.attr.name = "uevent";
  dev->uevent_attr.attr.mode = S_IWUSR;
  if (dev->driver)
       dev->uevent_attr.attr.owner = dev->driver->owner;
  dev->uevent_attr.store = store_uevent;
  device_create_file(dev, &dev->uevent_attr);
```

函数 device_add 第一部分要为设备在 sys 目录创建目录和 uevent 属性文件。这些内容已经多次出现了，就不一一分析了。

```
/* 设备的属性文件 */
if (MAJOR(dev->devt)) {
    struct device_attribute *attr;
    attr = kzalloc(sizeof(*attr), GFP_KERNEL);
    if (!attr) {
        error = -ENOMEM;
        goto PMError;
    }
    attr->attr.name = "dev";
    attr->attr.mode = S_IRUGO;
    if (dev->driver)
        attr->attr.owner = dev->driver->owner;
    attr->show = show_dev;
    error = device_create_file(dev, attr);
    if (error) {
        kfree(attr);
        goto attrError;
    }
    dev->devt_attr = attr;
}
/* 创建设备类的符号链接 *
if (dev->class) {
    sysfs_create_link(&dev->kobj, &dev->class->subsys.kset.kobj,
              "subsystem");
    sysfs_create_link(&dev->class->subsys.kset.kobj, &dev->kobj,
              dev->bus_id);

    sysfs_create_link(&dev->kobj, &dev->parent->kobj, "device");
    class_name = make_class_name(dev->class->name, &dev->kobj);
    sysfs_create_link(&dev->parent->kobj, &dev->kobj, class_name);
}
/* 设备的能源管理 */
if ((error = device_pm_add(dev)))
    goto PMError;
if ((error = bus_add_device(dev)))
    goto BusError;
kobject_uevent(&dev->kobj, KOBJ_ADD);
```

函数 device_add 第二部分要为设备创建一堆的符号链接和设备属性文件等。这仍是已经熟悉的内容，略过。

```
bus_attach_device(dev);
/* 设备加入父设备的链表 */
if (parent)
    klist_add_tail(&dev->knode_parent, &parent->klist_children);
```

函数 device_add 第三部分调用 bus_attach_device 把设备注册到总线，它的代码如代码清单 6-11 所示。

代码清单 6-11 bus_attach_device

```
void bus_attach_device(struct device * dev)
{
  struct bus_type * bus = dev->bus;

  if (bus) {
       device_attach(dev);
       klist_add_tail(&dev->knode_bus, &bus->klist_devices);
  }
}

int device_attach(struct device * dev)
{
  int ret = 0;

  down(&dev->sem);
  if (dev->driver) {
       /* 设备已经有了驱动，执行 sysfs 的链接和链表操作 */
       device_bind_driver(dev);
       ret = 1;
  } else
       ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
  up(&dev->sem);
  return ret;
}
```

回顾上一节添加驱动到总线的处理函数是 bus_for_each_dev，而本节添加设备到总线的处理函数是 bus_for_each_drv。前者的作用是遍历设备，为驱动寻找合适的设备，后者的作用是遍历驱动，为设备寻找合适的驱动。这两个函数的实现几乎一样，前者已经分析过，读者可以自行分析。

6.3 本章小结

本章分析了 platform 总线，我们发现和 input 架构的管理方式类似，总线下面也有两个列表，一个是设备列表，另一个是驱动列表。无论注册设备还是驱动都要遍历总线，寻找匹配的驱动或者设备。

platform 总线虽然简单但是很典型，基本说明了总线架构的概念。在内核驱动中，总线的使用很广泛，驱动目录里的 scsi、usb、ieee1394、pcmcia 等都是总线类型，它们也都使用了通用的设备和驱动管理。读者可以分析这些总线是怎么管理设备、怎么发现设备和管理驱动的，这样可以增强对设备管理架构的理解。

第 7 章 serio 总线

从第 5 章 input 驱动的分析中，我们了解到驱动可以分为几个层次，驱动之间可以嵌套。和这种架构类似，总线也可以分为几个层次，一种类型的总线可以架构在另一种类型的总线之上。

第 6 章 platform 总线驱动提供的 probe 函数中，调用了 serio_register_port 函数。这引出了总线嵌套的概念以及在内核中极重要的总线适配的概念。

7.1 什么是总线适配器

我们知道计算机的体系架构中，PCI 总线占有重要的地位，是连接 CPU 和外部设备的标准总线。而网卡、声卡、显卡、SCSI 卡等设备很多都是以 PCI 卡的形式出现，并插入计算机的 PCI 插槽。这些设备中，声卡显卡加载驱动后，就可以直接读写操作。而像 SCSI 卡这种设备就比较麻烦了，因为 SCSI 卡本身又可以连接 SCSI 硬盘，因此加载 SCSI 卡的 PCI 驱动后，必须进行 SCSI 总线扫描，发现 SCSI 硬盘设备，才能正确地读写硬盘。这里，SCSI 卡就担任了总线桥的任务，它提供了总线之间的协议转换和互操作。像 SCSI 卡这样的设备，称为**主机总线适配器**（HBA），它一方面是 PCI 设备，另一方面它又管理 SCSI 总线的设备。

7.2 向 serio 总线注册设备

第 6 章的例子中提到，注册到 platform 总线的设备和驱动匹配之后，驱动本身会探测端口并注册到 serio 总线。serio_register_port 函数就执行这个注册操作。serio 总线建筑在 platform 总线之上，它们分工合作，共同提供了完整的驱动功能。

从架构角度来看，serio 总线这种总线嵌套使用模式类似于总线适配器的模式，虽然从物理上来说，物理上存在的总线适配器和 serio 还是存在不同之处。

7.2.1 注册端口登记事件

我们接续第 6 章的分析，serio_register_port 函数的作用是注册 serio 总线，如代码清单 7-1 所示。

代码清单 7-1 serio_register_port 函数（serio.c）

```
static inline void serio_register_port(struct serio *serio)
{
  __serio_register_port(serio, THIS_MODULE);
}
void __serio_register_port(struct serio *serio, struct module *owner)
{
  serio_init_port(serio);
  /* 注册一个 SERIO_REGISTER_PORT 事件 */
  serio_queue_event(serio, owner, SERIO_REGISTER_PORT);
}
```

serio_register_port 函数的输入参数 serio 设置了端口类型是 SERIO_8042，说明是 8042 兼容型的（I8042 是 intel 开发的键盘控制芯片）。

serio_register_port 函数封装了 __serio_register_port 函数，后者首先初始化一个 serio 结构，设置总线类型为 serio 总线，然后调用 serio_queue_event 函数向系统注册一个端口登记事件。

serio_queue_event 函数作用是登记端口，如代码清单 7-2 所示。

代码清单 7-2 serio_queue_event(serio.c)

```
static void serio_queue_event(void *object, struct module *owner,
                enum serio_event_type event_type)
{
  unsigned long flags;
  struct serio_event *event;

  spin_lock_irqsave(&serio_event_lock, flags);

  list_for_each_entry_reverse(event, &serio_event_list, node) {
       /* 如果发现相同的 event，退出 */
       if (event->object == object) {
           if (event->type == event_type)
               goto out;
           break;
       }
  }

  if ((event = kmalloc(sizeof(struct serio_event), GFP_ATOMIC))) {
     ……/* 省略部分代码 */
      event->type = event_type;
      event->object = object;
      event->owner = owner;
```

```
        /*event 加到链表尾，并唤醒线程 */
        list_add_tail(&event->node, &serio_event_list);
        wake_up(&serio_wait);
```

serio_queue_event 函数首先遍历内核的 serio_event_list 链表，检查所有注册的事件，如果发现有相同类型的事件，直接退出。这说明同一个端口只能注册一次，如果重复登记，把它们合并为一次。然后创建一个 serio_event 结构，设置这个 serio_event 结构的类型为端口注册，唤醒处理这个事件的任务。

这个注册事件由谁来处理？实际是 serio_thread 内核线程处理的，如代码清单 7-3 所示。

代码清单 7-3 serio_thread (serio.c)

```
static int serio_thread(void *nothing)
{
  do {
        serio_handle_event();
        /* 内核线程进入睡眠状态。如果被唤醒，且事件链表非空则处理事件 */
        wait_event_interruptible(serio_wait,
            kthread_should_stop() || !list_empty(&serio_event_list));
        try_to_freeze();
  } while (!kthread_should_stop());

  printk(KERN_DEBUG "serio: kseriod exiting\n");
  return 0;
}
```

serio_thread 线程实际的处理由 serio_handle_event 函数执行，如代码清单 7-4 所示。

代码清单 7-4 serio_handle_event 函数

```
static void serio_handle_event(void)
{
  struct serio_event *event;

  mutex_lock(&serio_mutex);
  if ((event = serio_get_event())) {
        switch (event->type) {
              case SERIO_REGISTER_PORT:
                    serio_add_port(event->object);
                    break;
              default:
              break;
        }
```

serio_handle_event 函数处理各种事件，比如端口的注册和撤销、重新扫描端口等。对于 SERIO_REGISTER_PORT 事件，实际通过 serio_add_port 函数来处理，如代码清单 7-5 所示。

代码清单 7-5 serio_add_port (serio.c)

```
static void serio_add_port(struct serio *serio)
{
  int error;

  if (serio->parent) {
       /* 修改端口父设备的参数 */
       serio_pause_rx(serio->parent);
       serio->parent->child = serio;
       serio_continue_rx(serio->parent);
  }
    /* 把串口设备加入全局链表 */
  list_add_tail(&serio->node, &serio_list);
  if (serio->start)
       serio->start(serio);
  error = device_add(&serio->dev);
  if (error)
  ....../* 省略部分代码 */
  else {
       serio->registered = 1;
       error = sysfs_create_group(&serio->dev.kobj, &serio_id_attr_group);
```

serio_add_port 函数要调用 serio 结构的 start 函数，因为 q40kbd 注册端口的时候，并没有设置 start 函数，所以此处不会执行。

7.2.2 遍历总线的驱动

serio_add_port 函数的关键部分是 device_add 函数。在第 6 章 platform 总线的分析中，我们已经分析过这个函数，它的作用就是遍历总线的驱动，通过总线提供的 match 函数找到一个合适的驱动，然后调用总线的 probe 函数。

我们首先分析 serio 总线的 match 函数，然后再分析 probe 函数。

1. match 函数

serio 总线的 match 函数定义在 serio.c 文件，它的真实名字是 serio_bus_match，如代码清单 7-6 所示。

代码清单 7-6 serio_bus_match (serio.c)

```
static int serio_bus_match(struct device *dev, struct device_driver *drv)
{
  struct serio *serio = to_serio_port(dev);
  struct serio_driver *serio_drv = to_serio_driver(drv);
  /* 如果指定了手工绑定，则匹配不成功 */
  if (serio->manual_bind || serio_drv->manual_bind)
       return 0;

  return serio_match_port(serio_drv->id_table, serio);
}
```

serio_bus_match 函数在设备注册时多次调用，它的输入参数是 serio 总线上注册的每一个驱动，需要逐个检查端口设备 serio 和驱动的匹配情况。如果设备或者驱动设置了手工绑定，直接返回，否则调用 serio_match_port 函数检查设备和驱动的 ID 表是否匹配，如代码清单 7-7 所示。

代码清单 7-7　serio_match_port（serio.c）

```
static int serio_match_port(const struct serio_device_id *ids, struct serio *serio)
{
  while (ids->type || ids->proto) {
        if ((ids->type == SERIO_ANY || ids->type == serio->id.type) &&
           (ids->proto == SERIO_ANY || ids->proto == serio->id.proto) &&
           (ids->extra == SERIO_ANY || ids->extra == serio->id.extra) &&
           (ids->id == SERIO_ANY || ids->id == serio->id.id))
            return 1;
        ids++;
  }
  return 0;
}
```

serio_match_port 函数很简单，就是检查设备和驱动 ID 表的 type、proto 等参数是否相同。登记设备的时候，赋予的 type 是 SERIO_8042，搜索内核代码，和它匹配的驱动就是目录 drivers/input/keyboard 下的 atkbd.c 文件。

2. probe 函数

现在返回 device_add 函数，设备和驱动匹配之后，首先调用 serio 总线提供的 probe 函数，也就是 serio_driver_probe 函数，如代码清单 7-8 所示。

代码清单 7-8　serio_driver_probe（serio.c）

```
static int serio_driver_probe(struct device *dev)
{
  struct serio *serio = to_serio_port(dev);
  struct serio_driver *drv = to_serio_driver(dev->driver);

  return serio_connect_driver(serio, drv);
}
static int serio_connect_driver(struct serio *serio, struct serio_driver *drv)
{
  int retval;

  mutex_lock(&serio->drv_mutex);
  retval = drv->connect(serio, drv);
  mutex_unlock(&serio->drv_mutex);

  return retval;
}
```

serio_driver_probe 函数直接调用了 serio_connect_driver 函数，最终调用驱动提供的 connect 函数，就是 atkbd_connect，如代码清单 7-9 所示。

代码清单 7-9 atkbd_connect (atkbd.c)

```
static int atkbd_connect(struct serio *serio, struct serio_driver *drv)
{
  struct atkbd *atkbd;
  struct input_dev *dev;
  int err = -ENOMEM;

  atkbd = kzalloc(sizeof(struct atkbd), GFP_KERNEL);
  dev = input_allocate_device();
  if (!atkbd || !dev)
       goto fail;
  /*atkbd 的 dev 赋值为创建的 input 设备。input 要为这个设备加载对应的 input 驱动 */
  atkbd->dev = dev;
  /* 创建一个工作队列，这个队列做什么？就是处理 input_event 不处理的事件 */
  ps2_init(&atkbd->ps2dev, serio);
  INIT_WORK(&atkbd->event_work, atkbd_event_work, atkbd);
  mutex_init(&atkbd->event_mutex);

  switch (serio->id.type) {
       /* 对 8042_XL 芯片，设置 translated 成员为 1*/
       case SERIO_8042_XL:
           atkbd->translated = 1;
       case SERIO_8042:
           /* 如果赋值 write 函数，设置 write 为 1*/
           if (serio->write)
               atkbd->write = 1;
           break;
  }

  atkbd->softraw = atkbd_softraw;
  atkbd->softrepeat = atkbd_softrepeat;
  atkbd->scroll = atkbd_scroll;

  if (atkbd->softrepeat)
       atkbd->softraw = 1;

  serio_set_drvdata(serio, atkbd);
```

atkbd_connect 函数第一部分创建一个 input 设备和一个 atkbd 结构。atkbd 是 input 设备的控制结构，它封装了 input 设备的重要信息。

```
/* 打开 serio 登记的 open 函数。serio 在 q40kbd 里面创建的 */
err = serio_open(serio, drv);
if (err)
     goto fail;
```

```
if (atkbd->write) {
     if (atkbd_probe(atkbd)) {
         serio_close(serio);
         err = -ENODEV;
         goto fail;
     }
     atkbd->set = atkbd_select_set(atkbd, atkbd_set, atkbd_extra);
     atkbd_activate(atkbd);
} else {
     atkbd->set = 2;
     atkbd->id = 0xab00;
}
/* 根据 set 值选择码表 */
atkbd_set_keycode_table(atkbd);
/* 设置 input 设备的属性 */
atkbd_set_device_attrs(atkbd);
device_create_file(&serio->dev, &atkbd_attr_extra);
device_create_file(&serio->dev, &atkbd_attr_scroll);
device_create_file(&serio->dev, &atkbd_attr_set);
device_create_file(&serio->dev, &atkbd_attr_softrepeat);
device_create_file(&serio->dev, &atkbd_attr_softraw);
atkbd_enable(atkbd);
     input_register_device(atkbd->dev);
 return 0;
```

atkbd_connect 函数第二部分设置 atkbd 结构的属性和 input 设备的属性。因为 q40kbd 没有设置 write 函数，所以本场景设置 set 值为 2，然后要根据 set 值设置 atkbd 结构使用的码表。码表作用是把键盘输入设备的原始数据转换为统一的键值，供上层应用使用。

7.2.3 注册 input 设备

调用 input_register_device 注册 input 设备。这个函数在第 5 章分析过，它最终要为设备找到匹配的驱动。这个驱动就是键盘驱动，根据我们前面对 input 的分析，这个驱动通过 input_register_handler 函数加入到 input 的驱动列表。

atkbd_connect 函数开始部分调用了 serio 注册的 open 函数，这个函数就是 q40kbd_open，它的作用是打开键盘设备，以及设置设备的中断信息，如代码清单 7-10 所示。

代码清单 7-10 q40kbd_open

```
static int q40kbd_open(struct serio *port)
{
  q40kbd_flush();

  if (request_irq(Q40_IRQ_KEYBOARD, q40kbd_interrupt, 0, "q40kbd", NULL)) {
       printk(KERN_ERR "q40kbd.c: Can't get irq %d.\n", Q40_IRQ_KEYBOARD);
       return -EBUSY;
  }

  /* off we go */
```

```
    /* 写芯片的控制端口 */
    master_outb(-1, KEYBOARD_UNLOCK_REG);
    master_outb(1, KEY_IRQ_ENABLE_REG);
```

q40kbd_open 函数要申请中断并且设置设备端口。这个芯片是管理键盘的，设置芯片的 I/O 端口启动了键盘。而这里的中断是物理上存在的，它的中断处理函数需要调用 serio 设备、input 设备等一系列虚拟设备的处理函数，把来自物理底层的事件一步步报上去。

7.3 虚拟键盘驱动

上一节注册了一个 input 键盘设备，注册设备的同时需要找到它的驱动。

7.3.1 键盘驱动的初始化

input 作为一个设备框架，它提供的键盘驱动位于目录 drivers/char 下的 keyboard.c 文件。键盘驱动的初始化函数是 kbd_init，如代码清单 7-11 所示。

代码清单 7-11 kbd_init 函数（keyboard.c）

```
int __init kbd_init(void)
{
  int i;

       for (i = 0; i < MAX_NR_CONSOLES; i++) {
       kbd_table[i].ledflagstate = KBD_DEFLEDS;
       kbd_table[i].default_ledflagstate = KBD_DEFLEDS;
       kbd_table[i].ledmode = LED_SHOW_FLAGS;
       kbd_table[i].lockstate = KBD_DEFLOCK;
       kbd_table[i].slockstate = 0;
       kbd_table[i].modeflags = KBD_DEFMODE;
       kbd_table[i].kbdmode = VC_XLATE;
  }

  input_register_handler(&kbd_handler);
  /* 启动键盘的 tasklet*/
  tasklet_enable(&keyboard_tasklet);
  tasklet_schedule(&keyboard_tasklet);

  return 0;
}
```

kbd_init 函数设置 ID 表后，调用 input_register_handler 函数注册一个 input 驱动。

此时需要回顾 input 设备的注册过程，在设备匹配到驱动以后，还要调用驱动提供的 connect 函数和 start 函数。

7.3.2 与设备建立连接

首先我们从 connect 函数开始分析，这个函数的全名是 kbd_connect，作用是和具体的设备建立连接，执行打开设备的过程，如代码清单 7-12 所示。

代码清单 7-12 kbd_connect(keyboard.c)

```
static struct input_handle *kbd_connect(struct input_handler *handler,
                        struct input_dev *dev,
                        struct input_device_id *id)
{
  struct input_handle *handle;
  int i;
  /* 从保留键开始测试 */
  for (i = KEY_RESERVED; i < BTN_MISC; i++)
       if (test_bit(i, dev->keybit))
            break;

  if (i == BTN_MISC && !test_bit(EV_SND, dev->evbit))
       return NULL;

  handle = kzalloc(sizeof(struct input_handle), GFP_KERNEL);
  if (!handle)
       return NULL;

  handle->dev = dev;
  handle->handler = handler;
  handle->name = "kbd";

  input_open_device(handle);

  return handle;
}
```

atkbd_connect 函数创建一个 input_handle 结构，然后调用 input_open_device 函数执行 input 设备的 open 函数。对这个例子来说，因为设备没有注册 open 函数，实际上不会执行 open。

7.3.3 启动键盘设备

返回 input 设备的注册过程，驱动提供的 start 函数是 kbd_start，它的作用是点亮键盘的 LED 灯，启动键盘设备。如代码清单 7-13 所示。

代码清单 7-13 kbd_start(keyboard.c)

```
static void kbd_start(struct input_handle *handle)
{
  unsigned char leds = ledstate;
```

```
    tasklet_disable(&keyboard_tasklet);
    if (leds != 0xff) {
        input_inject_event(handle, EV_LED, LED_SCROLLL, !!(leds & 0x01));
        input_inject_event(handle, EV_LED, LED_NUML,    !!(leds & 0x02));
        input_inject_event(handle, EV_LED, LED_CAPSL,   !!(leds & 0x04));
        input_inject_event(handle, EV_SYN, SYN_REPORT, 0);
    }
    tasklet_enable(&keyboard_tasklet);
```

函数 kbd_start 开始控制真正的键盘设备，它的作用是刷新键盘的 LED 灯，这就是我们在系统启动时候看到的键盘闪烁。函数 input_inject_event 的作用是发送 LED 事件，让 LED 灯闪烁，这个功能必须由芯片来触发。根据目前学到的知识可以推测，这个过程一定要通过 input 调用 serio 总线上的设备驱动，写事件到相应的 I/O 端口。读者可以分析一下这个流程。

明白了原理和框架，其实内核的很多代码已经可以整理出大致的过程，只是具体实现的方法不同。因为我们选的是一个旧的键盘设备，这个芯片实际是不能驱动键盘闪烁的。

7.3.4 输入设备和主机系统之间的事件

输入设备和主机系统之间的事件有多种，这些事件类型如表 7-1 所示。

表 7-1 输入设备和主机系统之间的事件

事　件	功能描述	事　件	功能描述
EV_SYN	同步	EV_LED	LED 灯事件
EV_KEY	按键	EV_SND	声音事件
EV_REL	相对坐标	EV_REP	自动重复的参数事件
EV_ABS	绝对坐标	EV_SW	切换事件
		EV_MSC	其他杂项事件

这些事件类型很多，每种事件里面又定义了很多子事件。针对具体事件的处理，需要相关驱动的开发人员仔细设计。

7.4 键盘中断

现在总结设备和驱动处理的整个路径。最底层的设备是 Q40kbd，它是一个 platform 总线设备，和 platform 总线的驱动匹配后，注册了一个 serio 端口设备。serio 设备也需要匹配 serio 总线的驱动 atkbd，然后调用驱动的 connect 函数创建并注册一个 input 设备。input 设备再次匹配 input 之上登记的驱动，最终找到 input 驱动 kbd。

通过一层层的总线、设备和驱动的搜索和匹配，内核最终建立了设备的驱动架构。一旦设备和驱动都就位了，那么就可以从键盘接收用户的输入了。

7.4.1 q40kbd 设备的中断处理

最底层的设备是 q40kbd，当用户按键的时候，触发它的中断，然后系统将一层层往上报按键事件，所以需要从 q40kbd 设备的中断处理函数开始分析。中断处理函数 q40kbd_interrupt 如代码清单 7-14 所示。

代码清单 7-14 q40kbd_interrupt(q40kbd.c)

```
static irqreturn_t q40kbd_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  unsigned long flags;
  spin_lock_irqsave(&q40kbd_lock, flags);

  if (Q40_IRQ_KEYB_MASK & master_inb(INTERRUPT_REG))
      serio_interrupt(q40kbd_port, master_inb(KEYCODE_REG), 0, regs);
 /* 向 I/O 端口 KEYBOARD_UNLOCK_REG 写数据 */
  master_outb(-1, KEYBOARD_UNLOCK_REG);
  spin_unlock_irqrestore(&q40kbd_lock, flags);

  return IRQ_HANDLED;
}
```

q40kbd_interrupt 读 I/O 端口 KEYCODE_REG 的数据，作为输入参数调用 serio_interrupt 函数来处理。

7.4.2 serio 总线的中断处理

serio_interrupt 是 serio 总线提供的中断处理函数，它要进一步调用驱动提供的中断处理函数，如代码清单 7-15 所示。

代码清单 7-15 serio_interrupt（serio.c）

```
irqreturn_t serio_interrupt(struct serio *serio,
      unsigned char data, unsigned int dfl, struct pt_regs *regs)
{
     ……/* 省略部分代码 */
     if (likely(serio->drv)) {
         ret = serio->drv->interrupt(serio, data, dfl, regs);
  } else if (!dfl && serio->registered) {
      /*serio 重新扫描，和前面的设备注册差不多的流程 */
      serio_rescan(serio);
      ret = IRQ_HANDLED;
  }
```

函数 serio_interrupt 检查是否已经匹配驱动，如果是，调用驱动的中断处理函数。

7.4.3 驱动提供的中断处理

serio 驱动是什么？就是前一节分析的 atkbd，所以中断处理函数就是 atkbd_interrupt，如

代码清单 7-16 所示。

代码清单 7-16 atkbd_interrupt（atkbd.c）

```
static irqreturn_t atkbd_interrupt(struct serio *serio, unsigned char data,
            unsigned int flags, struct pt_regs *regs)
{
  ....../* 省略变量定义代码 */
 /* 如果需要，回一个 ACk 给键盘 */
  if (unlikely(atkbd->ps2dev.flags & PS2_FLAG_ACK))
      if  (ps2_handle_ack(&atkbd->ps2dev, data))
           goto out;
 /* 如果有进程在等键盘，唤醒等待的进程 */
  if (unlikely(atkbd->ps2dev.flags & PS2_FLAG_CMD))
      if  (ps2_handle_response(&atkbd->ps2dev, data))
           goto out;

  if (!atkbd->enabled)
      goto out;
/* 上报原始的扫描码 */
input_event(dev, EV_MSC, MSC_RAW, code);
      ....../* 省略部分代码 */

   /* 转换扫描码为通用的字符码，码表就是 atkbd_connect 时注册进去的码表 */
  keycode = atkbd->keycode[code];

  if (keycode != ATKBD_KEY_NULL)
      input_event(dev, EV_MSC, MSC_SCAN, code);
```

函数 atkbd_interrupt 要三次调用 input_event 上报输入的数据：第一次上报原始的扫描码；第二次也是上报原始的扫描码，只是为了兼容性考虑，对扫描码做了处理；第三次上报用户真正需要的按键值。

```
input_regs(dev, regs);
      input_event(dev, EV_KEY, keycode, value);
      /* 同步事件，上报结束 */
input_sync(dev);

if (value && add_release_event) {
               input_report_key(dev, keycode, 0);
               input_sync(dev);
          }
      }
      ....../* 省略部分代码 */
```

函数 atkbd_interrupt 第二部分上报真正的按键值。按键值根据设备的转换码表将原始的扫描码转换后得到，对上层应用来说，按键值是统一的。

上报输入事件通过函数 input_event 完成，它要处理同步事件、按键事件、绝对坐标、LED 灯、声音等各种事件，而我们最关心的是按键事件，所以省略了其他事件的处理代码，

如代码清单 7-17 所示。

代码清单 7-17 input_event（input.c）

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)
{
  struct input_handle *handle;

  if (type > EV_MAX || !test_bit(type, dev->evbit))
      return;
  /* 产生随机数，这里利用按键产生随机数 */
  add_input_randomness(type, code, value);

  switch (type) {
      /*EV_KEY 是按键事件 */
      case EV_KEY:
          if (code > KEY_MAX || !test_bit(code, dev->keybit) ||
            !!test_bit(code, dev->key) == value)
              return;

           if (value == 2)
              break;

           change_bit(code, dev->key);

           if (test_bit(EV_REP, dev->evbit) && dev->rep[REP_PERIOD] &&
                dev->rep[REP_DELAY] && dev->timer.data && value) {
                dev->repeat_key = code;
                mod_timer(&dev->timer, jiffies +
                          msecs_to_jiffies(dev->rep[REP_DELAY]));
           }
           break;
```

input_event 函数的第一部分是对各种输入事件进行处理。对按键事件而言，如果设备要求周期重复上报按键，要启动 input 设备的定时器，启动时间为当前时间加上设备的延时时间。当启动时间到达后，重复上报按键值。

```
if (dev->grab)
    dev->grab->handler->event(dev->grab, type, code, value);
else
    list_for_each_entry(handle, &dev->h_list, d_node)
         if (handle->open)
             handle->handler->event(handle, type, code, value);
```

input_event 函数的第二部分调用 input 驱动对输入事件进行处理。如果设备已经有了指定的 input_handler，也就是有了指定的驱动，则调用驱动的 event 函数，否则遍历设备的 input_handler 链表，逐个调用驱动的 event 函数。用户的按键经过层层的驱动到最后，终于汇入了 input 字符设备定义的架构，此时原始的按键数据已经变成了标准的输入键值。

input 键盘设备驱动的 event 函数其实就是 kbd_event，它的代码如代码清单 7-18 所示。

代码清单 7-18 kbd_event(keyboard.c)

```
static void kbd_event(struct input_handle *handle, unsigned int event_type,
          unsigned int event_code, int value)
{
  /* 原始码由 kbd_rawcode 处理 */
  if (event_type == EV_MSC && event_code == MSC_RAW && HW_RAW(handle->dev))
      kbd_rawcode(value);
  /* 标准的字符码，由 kbd_keycode 处理 */
  if (event_type == EV_KEY)
      kbd_keycode(event_code, value, HW_RAW(handle->dev), handle->dev->regs);
  tasklet_schedule(&keyboard_tasklet);
  do_poke_blanked_console = 1;
  schedule_console_callback();
```

kbd_event 根据上报输入事件的类型分别处理，如果上报原始扫描码，调用 kbd_rawcode 函数处理，如果是经过转换的键值，则调用 kbd_keycode 函数处理。这两个函数最终都要调用 put_queue 函数把按键数据送到控制台的输入缓冲区里。

控制台是 Linux 输入 / 输出系统的一个重要概念。控制台简单地把用户的输入发送到计算机处理，然后再把处理结果返回给用户。从软件角度看，控制台提供给用户一个使用命令行的字符界面，用于接收用户输入和反馈输出结果。由于现代计算机功能强大，可以利用硬件模拟出来很多控制台界面，Linux 系统支持多个控制台，而用户的输入由当前活跃的控制台接收。kbd_event 把用户的按键数据送到当前控制台的输入缓冲区，应用程序调用标准的库函数 scanf 读控制台的输入缓冲区，就可以获得用户的按键值。

这里启用了一个 tasklet_schedule，处理 led 事件等耗时间的操作，用软中断继续处理。回顾基础知识一节，软中断的好处是软中断里面可以打开中断。所以一些和硬件相关的操作放在中断里面，而和硬件无关的逻辑等放在软中断，可以减少长时间关中断的代价（中断的处理代码也可以打开中断，软中断处理也可能需要关闭中断，是否开闭中断，需要仔细设计）。

7.5 本章小结

platform 总线和 serio 总线构成了总线的嵌套关系。这个例子比较简单，但是却很典型。对于一个真正的计算机系统来说，最重要的总线是 PCI 总线，大多数设备都是 PCI 设备，而总线适配器（HBA）一般都是挂载到 PCI 总线上。作为最重要的总线，下章将分析 PCI 总线。

第 8 章

PCI 总线

前面的章节先后介绍了 input 字符设备以及 platform 总线。input 设备是个逻辑概念，它建立在真正的物理设备之上，是对设备功能的抽象表达。本章要重点分析真实的物理设备和 PCI 总线的应用。

8.1 深入理解 PCI 总线

PCI 总线是现代计算机系统中最重要的总线，当 PCI 总线扫描到 PCI 设备后，已经为设备设置了 DMA 信息、中断信息和 I/O 端口、I/O 内存信息，这些信息是实现 PCI 设备驱动的基础，也是深入理解 PCI 设备驱动的重点。

8.1.1 PCI 设备工作原理

PCI 设备具有自己的设备配置信息，也具备 I/O 端口和 I/O 内存，这些端口和内存构成一个独立的地址空间，就是 PCI 总线地址空间。这个空间和主存的空间是隔离的。彼此互相独立，CPU 要通过主桥（host bridge）才能访问 PCI 地址空间，而 PCI 设备也要通过主桥才能访问主存。

主桥可以直接产生一条 PCI 总线，这条总线也是主桥管理的第一条总线，也是 0 号 PCI 总线。在内核代码中，会直接使用这条 0 号总线。从该总线还可以扩展出一系列的 PCI 总线，称为 PCI 桥，以主桥为根节点，这些桥和设备形成了一颗 PCI 树。这些扩展出来的 PCI 总线都可以连接 PCI 设备，但是在一条 PCI 总线上，最多只能挂载 256 个 PCI 设备（PCI 桥本身也是一个 PCI 设备）。

如图 8-1 所示，PCI 0 号总线下面有四个 PCI 设备，其中一个是桥设备，这个桥设备引出了 PCI 总线 1，它的下面又可以挂载 256 个 PCI 设备。

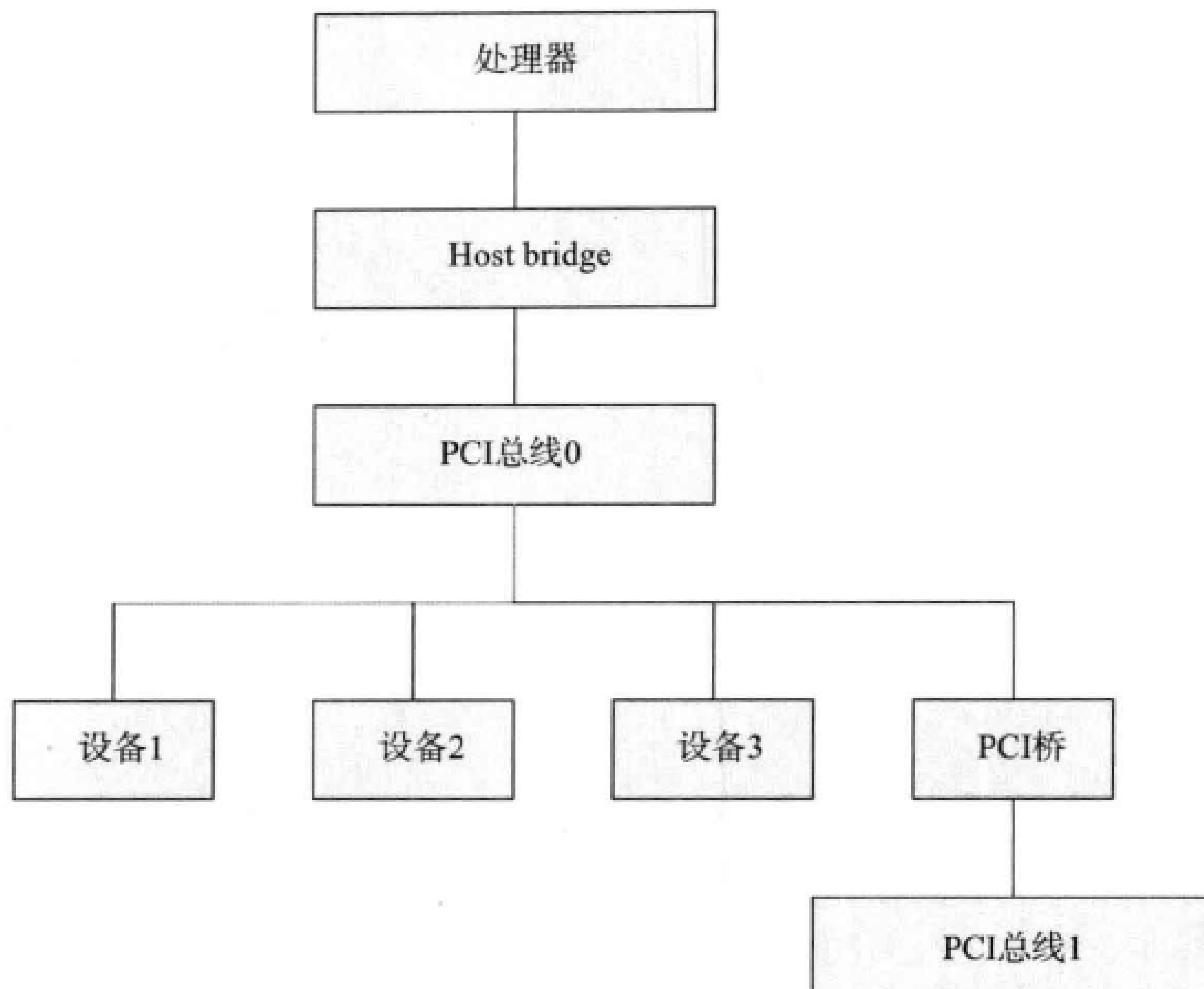


图 8-1 PCI 设备和总线图

8.1.2 PCI 总线域

PCI 设备具有一个 8 bit 的总线号，一个 5 bit 的设备编号以及一个 3 bit 的功能编号。

因此一个主桥下最多拥有 256 个总线，这个对大型系统上而言是不够的，为此 Linux 引入 PCI 域的概念。每个 PCI 域可拥有 256 个总线，而每个总线可有 32 个设备。如果设备是多功能设备，还可以支持 8 个子设备。图 8-2 给出了一个简单系统的 PCI 设备图。

```
bash-4.1$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 01)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Unassigned class [ff80]: XenSource, Inc. Xen Platform Device (rev 01)
bash-4.1$
```

图 8-2 PCI 设备图

以图 8-2 显示的 PCI 设备 00:01:0 为例，分隔符将设备地址分为三个区间，首区间 00 是总线号，中间区间 01 是设备编号，末尾区间 0 是功能编号，这是一个多功能 PCI 设备。

8.1.3 PCI 资源管理

为了管理 PCI 设备的 I/O 端口和 I/O 内存，内核定义了一个 resource 结构。

首先分析代表 I/O 端口的 resource，如下所示：

```
struct resource ioport_resource = {
  .name     = "PCI IO",
  .start    = 0,
  .end      = IO_SPACE_LIMIT,
  .flags    = IORESOURCE_IO,
};
```

ioport_resource 起始地址为 0，结束地址为 0xffff，这个变量定义了全部的 I/O 端口地址空间，每个 PCI 设备新加入系统，都要检查它配置空间的 I/O 端口的结束地址是否大于 0xffff，和现存设备是否有冲突，是否可以插入 I/O 端口地址空间。

I/O 内存则是另一个 resource 结构，如下所示：

```
struct resource iomem_resource = {
  .name     = "PCI mem",
  .start    = 0,
  .end      = -1,
  .flags    = IORESOURCE_MEM,
};
```

I/O 内存需要映射到主机内存地址空间，所以它的结束地址为 –1。当 PCI 设备加入系统的时候，同样要检查它配置空间的 I/O 内存和其他设备是否有冲突，是否可以插入 I/O 内存地址空间。

8.1.4 PCI 配置空间读取和设置

第 3 章已经介绍了，对 PCI 设备的读取和设置要通过 I/O 指令读一个特殊的 I/O 端口空间来完成。内核中提供了一个 pci_raw_ops 结构来控制配置空间的读写。这个结构的读写函数通常被设置为 pci_conf1_read 函数和 pci_conf1_write 函数，通过它们执行对 PCI 配置空间的读写。

有必要分析 pci_conf1_read 函数对 PCI 配置空间的读取过程，如代码清单 8-1 所示。

代码清单 8-1 pci_conf1_read (direct.c)

```
int pci_conf1_read(unsigned int seg, unsigned int bus,
             unsigned int devfn, int reg, int len, u32 *value)
{
  ....../* 省略部分代码 */
  outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);

  switch (len) {
  case 1:
       *value = inb(0xCFC + (reg & 3));
       break;
  case 2:
       *value = inw(0xCFC + (reg & 2));
       break;
  case 4:
       *value = inl(0xCFC);
```

```
        break;
    }
```

pci_conf1_read 函数首先往 I/O 端口 0xCF8 写入一个 PCI 地址，然后从 I/O 端口 0xCFC 读该地址的配置信息。pci_conf1_write 函数和 pci_conf1_read 函数很类似，此处不再分析。

8.2 PCI 设备扫描过程

Linux 内核具备多种 PCI 的扫描方式，它们彼此之间大同小异。作为例子，本节选择典型的传统扫描模式，这种扫描模式定义在 legacy.c 文件。

传统扫描模式的执行函数是 pci_legacy_init，如代码清单 8-2 所示。

代码清单 8-2 pci_legacy_init 函数（legacy.c）

```
static int __init pci_legacy_init(void)
{
  if (!raw_pci_ops) {
       printk("PCI: System does not support PCI\n");
       return 0;
  }
  /* 如果曾经扫描过，则不再扫描 */
  if (pcibios_scanned++)
       return 0;

  printk("PCI: Probing PCI hardware\n");
  /* 扫描 0 号总线 */
  pci_root_bus = pcibios_scan_root(0);
  if (pci_root_bus)/* 扫描出来的设备加入 pci 总线 */
       pci_bus_add_devices(pci_root_bus);
  /* 对 bios 提供的 PCI 总线进行扫描 */
  pcibios_fixup_peer_bridges();

   return 0;
}
```

pci_legacy_init 函数首先扫描 0 号总线，如果扫描成功，则把 0 号总线作为系统的根总线。然后，要把 0 号总线上扫描出来的所有设备都加入一个全局的 PCI 设备链表。最后，调用 pcibios_fixup_peer_bridges 对 bios 提供的 PCI 总线做进一步的扫描。

8.2.1 扫描 0 号总线

扫描 0 号总线调用的是 pcibios_scan_root 函数，代码如代码清单 8-3 所示。

代码清单 8-3 pcibios_scan_root 函数

```
struct pci_bus * __devinit pcibios_scan_root(int busnum)
{
```

```
  struct pci_bus *bus = NULL;

  dmi_check_system(pciprobe_dmi_table);
  while ((bus = pci_find_next_bus(bus)) != NULL) {
       if (bus->number == busnum) {
           /* Already scanned */
           return bus;
       }
  }
  printk(KERN_DEBUG "PCI: Probing PCI hardware (bus %02x)\n", busnum);
  return pci_scan_bus_parented(NULL, busnum, &pci_root_ops, NULL);
}
```

pcibios_scan_root 函数首先逐个遍历所有的 PCI 总线，检查指定的总线是否已经扫描过，如果已经扫描，则直接返回。如果尚未扫描，则调用 pci_scan_bus_parented 函数扫描总线。

8.2.2 扫描总线上的 PCI 设备

pci_scan_bus_parented 函数的功能是扫描总线上可能接入的 256 个 PCI 设备，如果扫描到的 PCI 设备是个桥设备，还要递归扫描桥设备，把桥设备可能接入的 PCI 设备扫描出来，这个函数的代码清单 8-4 所示。

代码清单 8-4 pci_scan_bus_parented 函数

```
struct pci_bus * __devinit pci_scan_bus_parented(struct device *parent,
       int bus, struct pci_ops *ops, void *sysdata)
{
  struct pci_bus *b;
  /* 创建一个总线对象 */
  b = pci_create_bus(parent, bus, ops, sysdata);
  if (b)
       b->subordinate = pci_scan_child_bus(b);
  return b;
}
```

pci_scan_bus_parented 函数可分成两个步骤：第一步是创建一个总线对象，第二步是调用 pci_scan_child_bus 对创建的总线对象进行递归扫描。

1. 创建一个总线对象

首先分析创建总线对象的 pci_create_bus 函数，它的 parent 参数为空，说明这条总线没有父设备，是一条根总线。

1）pci_create_bus 第一部分是创建一个总线对象和一个设备对象。

它的代码如代码清单 8-5 所示。

代码清单 8-5 pci_create_bus 函数

```
struct pci_bus * __devinit pci_create_bus(struct device *parent,
       int bus, struct pci_ops *ops, void *sysdata)
```

```
{
  int error;
  struct pci_bus *b;
  struct device *dev;
  /* 申请一个 PCI 总线结构 */
  b = pci_alloc_bus();
  if (!b)
       return NULL;
  /* 申请一个 dev 结构 */
  dev = kmalloc(sizeof(*dev), GFP_KERNEL);
  ……/* 省略部分代码 */

  b->sysdata = sysdata;
  b->ops = ops;
  /* 检查是否被创建 */
  if (pci_find_bus(pci_domain_nr(b), bus)) {
       /* If we already got to this bus through a different bridge, ignore it */
       pr_debug("PCI: Bus %04x:%02x already known\n", pci_domain_nr(b), bus);
       goto err_out;
  }
  /* 创建的总线加入 PCI 总线链表 */
  down_write(&pci_bus_sem);
  list_add_tail(&b->node, &pci_root_buses);
  up_write(&pci_bus_sem);
```

PCI总线本身就是一个设备，所以除了总线对象外，还要为它创建一个设备对象。总线对象要链接到一个全局的链表头pci_root_buses，这样通过这条链表，就可以遍历所有的PCI总线。

2）pci_create_bus函数第二部分执行结构和对象的注册。

```
/* 设置 dev 结构并登记到系统 */
memset(dev, 0, sizeof(*dev));
dev->parent = parent;
dev->release = pci_release_bus_bridge_dev;
sprintf(dev->bus_id, "pci%04x:%02x", pci_domain_nr(b), bus);
error = device_register(dev);
if (error)
     goto dev_reg_err;
b->bridge = get_device(dev);

b->class_dev.class = &pcibus_class;
sprintf(b->class_dev.class_id, "%04x:%02x", pci_domain_nr(b), bus);
error = class_device_register(&b->class_dev);
if (error)
     goto class_dev_reg_err;
error = class_device_create_file(&b->class_dev, &class_device_attr_cpuaffinity);
if (error)
     goto class_dev_create_file_err;
```

```
/* Create legacy_io and legacy_mem files for this bus */
pci_create_legacy_files(b);

error = sysfs_create_link(&b->class_dev.kobj, &b->bridge->kobj, "bridge");
if (error)
     goto sys_create_link_err;

b->number = b->secondary = bus;
```

首先把设备对象注册到系统，这个过程在第 6 章已经分析过了。其次是注册 PCI 总线类和为 sysfs 文件系统创建符号连接。

3）pci_create_bus 函数最后设置 PCI 总线的资源。

```
/* 设置总线的资源 */
b->resource[0] = &ioport_resource;
b->resource[1] = &iomem_resource;
return b;
```

PCI 总线的资源有两类，一类是 I/O 端口，另一类是 I/O 内存。总线上所有设备的端口和内存组成了一个空间，为了避免冲突，内核设置了全局的数据结构 ioport_resource 和 iomem_resource，分别保存所有的 I/O 端口资源和所有的 I/O 内存资源。

2. 扫描总线

现在返回 pci_scan_bus_parented 函数，当成功创建总线对象后，开始扫描这条总线。扫描总线调用 pci_scan_child_bus 函数，如代码清单 8-6 所示。

代码清单 8-6　pci_scan_child_bus（probe.c）

```
unsigned int __devinit pci_scan_child_bus(struct pci_bus *bus)
{
  unsigned int devfn, pass, max = bus->secondary;
  struct pci_dev *dev;

  pr_debug("PCI: Scanning bus %04x:%02x\n", pci_domain_nr(bus), bus->number);

  /* Go find them, Rover! */ /* 扫描总线下面的 256 个设备 */
  for (devfn = 0; devfn < 0x100; devfn += 8)
       pci_scan_slot(bus, devfn);

  /*
   * After performing arch-dependent fixup of the bus, look behind
   * all PCI-to-PCI bridges on this bus.
   */
  pr_debug("PCI: Fixups for bus %04x:%02x\n", pci_domain_nr(bus), bus->number);
  pcibios_fixup_bus(bus);
  /* 扫描子总线，分两次扫描，第一次是扫描 bios 发现的总线 */
  for (pass=0; pass < 2; pass++)
       list_for_each_entry(dev, &bus->devices, bus_list) {
            if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
                dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
```

```
                max = pci_scan_bridge(bus, dev, max, pass);
        }
```

扫描 PCI 总线是个递归的过程。每条 PCI 总线可以配置 32 个多功能设备，每个多功能设备又可以安装 8 个子设备，总共就是 256 个设备。这 256 个设备中，有的设备可能是 PCI 桥，每个 PCI 桥下面又可以接入 256 个设备。通过函数 pci_scan_slot 扫描每个多功能设备的 8 个子设备，通过 pci_scan_bridge 函数扫描 PCI 桥设备。对于桥设备，还要递归调用 pci_scan_child_bus 函数扫描本桥设备下面可能接入的 PCI 设备。

8.2.3 扫描多功能设备

扫描 PCI 多功能设备和扫描桥设备有重复的地方，因此本节以扫描多功能设备的函数 pci_scan_slot 为例进行分析，它的代码如代码清单 8-7 所示。

代码清单 8-7 pci_scan_slot 函数（probe.c）

```
int __devinit pci_scan_slot(struct pci_bus *bus, int devfn)
{
  int func, nr = 0;
  int scan_all_fns;

  scan_all_fns = pcibios_scan_all_fns(bus, devfn);

  for (func = 0; func < 8; func++, devfn++) {
       struct pci_dev *dev;
       dev = pci_scan_single_device(bus, devfn);
       if (dev) {
           nr++;
           /*
            * If this is a single function device,
            * don't scan past the first function.
           if (!dev->multifunction) {
               if (func > 0) {
                   dev->multifunction = 1;
               } else {
                   break;
               }
           }
       } else {
           if (func == 0 && !scan_all_fns)
               break;
       }
  }
  return nr;
}
```

pci_scan_slot 函数从 0 号设备开始进行扫描，如果扫描发现是单功能设备，不再继续扫描，如果发现是多功能设备，则进行 8 次扫描。

8.2.4 扫描单个设备

扫描单个设备调用 pci_scan_single_device 函数，输入参数是总线结构和设备功能号，如代码清单 8-8 所示。

代码清单 8-8　pci_scan_single_device 函数（probe.c）

```
pci_scan_single_device(struct pci_bus *bus, int devfn)
{
  struct pci_dev *dev;

  dev = pci_scan_device(bus, devfn);
  if (!dev)
       return NULL;

  pci_device_add(dev, bus);
  pci_scan_msi_device(dev);

  return dev;
}
```

pci_scan_single_device 函数调用 pci_scan_device 扫描设备，扫描成功后把设备加入总线的设备链表。最后的 pci_scan_msi_device 函数是检查设备的 MSI 能力，MSI 和设备的中断有关，当前可以不关心。

8.2.5 扫描设备信息

扫描 PCI 设备通过读取 PCI 设备的配置空间完成，这部分原理在第 3 章已经介绍过。扫描设备的代码在 pci_scan_device 函数中，如代码清单 8-9 所示。

代码清单 8-9　pci_scan_device（probe.c）

```
pci_scan_device(struct pci_bus *bus, int devfn)
{
  struct pci_dev *dev;
  u32 l;
  u8 hdr_type;
  int delay = 1;
  /* 读 PCI 设备制造商的 ID*/
  if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
       return NULL;

  /* some broken boards return 0 or ~0 if a slot is empty: */
  if (l == 0xffffffff || l == 0x00000000 ||
       l == 0x0000ffff || l == 0xffff0000)
       return NULL;

  /* 处理需要重复读配置信息的情况 */
  while (l == 0xffff0001) {
       msleep(delay);
```

```
        delay *= 2;
        if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
            return NULL;
        /* Card hasn't responded in 60 seconds?  Must be stuck. */
        if (delay > 60 * 1000) {
            printk(KERN_WARNING "Device %04x:%02x:%02x.%d not "
                    "responding\n", pci_domain_nr(bus),
                    bus->number, PCI_SLOT(devfn),
                    PCI_FUNC(devfn));
            return NULL;
        }
    }
```

pci_scan_device 函数第一部分读 PCI 设备制造商的 ID，所有的制造商都要分配厂商 ID 号，从 ID 就可以获得设备厂商信息。这部分代码要处理异常情况，某些设备可能返回重试状态，这种情况要延迟一段时间，再次尝试读制造商的 ID，如果延迟时间超过 60 秒，还没有读到 ID，则返回失败。

pci_scan_device 函数第二部分为设备分配一个 PCI 设备结构，然后根据设备配置空间读取的信息对设备进行赋值。

```
    /* 读 PCI 设备的类型 */
    if (pci_bus_read_config_byte(bus, devfn, PCI_HEADER_TYPE, &hdr_type))
        return NULL;
    /* 申请一个 PCI 设备结构 */
    dev = kzalloc(sizeof(struct pci_dev), GFP_KERNEL);
    if (!dev)
        return NULL;
    /* 设置 PCI 设备的参数，包括类型、制造商和是否多功能等 */
    dev->bus = bus;
    dev->sysdata = bus->sysdata;
    dev->dev.parent = bus->bridge;
    dev->dev.bus = &pci_bus_type;
    dev->devfn = devfn;
    dev->hdr_type = hdr_type & 0x7f;
    dev->multifunction = !!(hdr_type & 0x80);
    dev->vendor = l & 0xffff;
    dev->device = (l >> 16) & 0xffff;
    dev->cfg_size = pci_cfg_space_size(dev);
    dev->error_state = pci_channel_io_normal;

    /* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
       set this higher, assuming the system even supports it.  */
    /* 设置设备的 dma 地址掩码 */
    dev->dma_mask = 0xffffffff;
    if (pci_setup_device(dev) < 0) {
        kfree(dev);
        return NULL;
    }
```

```
  return dev;
}
```

此时，只读取了配置空间的制造商 ID 和头部信息（HEADER_TYPE），信息的进一步读取在函数 pci_setup_device 中完成，这个函数同时要设置 PCI 设备的信息，如代码清单 8-10 所示。

代码清单 8-10 pci_setup_device

```
static int pci_setup_device(struct pci_dev * dev)
{
  u32 class;

  sprintf(pci_name(dev), "%04x:%02x:%02x.%d", pci_domain_nr(dev->bus),
       dev->bus->number, PCI_SLOT(dev->devfn), PCI_FUNC(dev->devfn));
   /* 读类别 */
  pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
  class >>= 8;                                /* upper 3 bytes */
  dev->class = class;
  class >>= 8;
  /* "Unknown power state" */
  dev->current_state = PCI_UNKNOWN;

  /* Early fixups, before probing the BARs */
  pci_fixup_device(pci_fixup_early, dev);
  class = dev->class >> 8;
```

pci_setup_device 函数第一部分是读设备类的信息。前面的高 24 位是 class 信息，后面的低 8 位是 revision 信息，并根据读取的信息设置 PCI 设备。

pci_setup_device 函数第二部分是根据设备的类型读需要的信息。

```
switch (dev->hdr_type) {                    /* header type */
  case PCI_HEADER_TYPE_NORMAL:              /* standard header */
       if (class == PCI_CLASS_BRIDGE_PCI)
            goto bad;
       /* 读中断信息 */
       pci_read_irq(dev);
       /* 读配置空间的资源信息，总共可以有 6 条信息 */
       pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
       /* 读子系统厂商 ID*/
       pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
       /* 读子系统 ID*/
       pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);
       break;

  case PCI_HEADER_TYPE_BRIDGE:              /* bridge header */
       if (class != PCI_CLASS_BRIDGE_PCI)
            goto bad;
       /* The PCI-to-PCI bridge spec requires that subtractive
          decoding (i.e. transparent) bridge must have programming
```

```
        interface code of 0x01. */
      pci_read_irq(dev);
      dev->transparent = ((dev->class & 0xff) == 1);
      pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
      break;
……/* 省略 card bus 类型的处理代码 */
```

设备有三种类型：通常的 PCI 设备、PCI 桥设备和 CARDBUS 设备。每种设备都要读中断信息和资源信息。PCI 设备的配置空间提供两种资源，一种是 I/O 端口，另一种是 I/O 内存。普通设备可以提供六个资源信息，而桥设备只有两个资源信息。

操作系统扫描 PCI 总线，目的就是获得 PCI 设备的信息，然后为每个设备分配一个 PCI 设备结构。PCI 总线扫描到设备之后，需要为设备加载正确的驱动。这部分内容在第 6 章和第 7 章已经介绍，此处不再分析。

8.3 本章小结

PCI 总线可说是现代计算机系统中最重要的总线，当 PCI 总线扫描到 PCI 设备后，已经为设备设置了 DMA 信息、中断信息和 I/O 端口、I/O 内存信息，这些信息是实现 PCI 设备驱动的基础，也是深入理解 PCI 设备驱动的重要点。内核中实现了大量 PCI 设备的驱动，读者可以挑选熟悉的驱动，分析一下驱动如何处理这些信息。

第 9 章
块设备

对于驱动工程师来说，块设备和字符设备是开发过程中经常用到的概念。字符设备通过函数 init_special_inode 为字符设备设置函数指针。对于块设备而言，这部分的架构是相同的，也是通过 init_special_inode 为块设备设置函数指针。不同之处是，赋予字符设备的函数指针结构是 def_chr_fops，而赋予块设备的函数指针结构是 def_blk_fops。

这种类似的架构减小了学习的理解难度，能从已知的知识点推广到未知的知识点，可以提升学习的信心。

9.1 块设备的架构

和字符设备比较，块设备有很多不同的地方。实际上，块设备常常和磁盘关联在一起，它的使用和管理比字符设备要复杂。本章的分析忽略块设备和字符设备相同的地方，重点介绍块设备的独特之处。首先从块设备的结构定义着手。

9.1.1 块设备、磁盘对象和队列

块设备一般总和通用磁盘对象 gendisk 捆绑在一起。块设备的结构定义如下所示，其中省略了当前不关心的内容：

```
struct block_device {
  ……/* 省略部分代码 */
  struct gendisk *bd_disk;
}
```

通用磁盘对象是在计算机启动时扫描磁盘或者磁盘插入计算机槽位时，内核为物理磁盘创建的数据结构（光盘、磁带设备也用通用磁盘对象表示）。

通用磁盘对象创建后，一般要在根目录的 dev 目录下面，创建一个设备文件，这个设备文件被指明为块设备，具有自己的磁盘名。所以通用磁盘对象创建在前，等用户打开块设备时，会绑定块设备到相关的通用磁盘对象。

通用磁盘对象的结构定义如下：

```
struct gendisk {
  int major;              /* major number of driver */
  int first_minor;
  int minors; /* maximum number of minors, =1 for disks that can't be partitioned. */
  char disk_name[32];     /* name of major driver */
  struct block_device_operations *fops;
  struct request_queue *queue;
  ……/* 省略部分代码 */
}
```

结构定义中省略了一些无关的成员，保留重要的成员。结构成员首先是主从设备号，然后是磁盘的名字。

区别块设备和字符设备的最重要成员就是队列 queue，所有对通用磁盘对象的 I/O 操作都要进入这个队列 queue 排队，然后再由内核处理。这里块设备队列是个笼统的说法，其实块设备使用的队列既包括块设备自身的队列，也包括块设备隐含的电梯对象的队列。在具体的使用中，可以看到这两种队列的不同之处。

9.1.2 块设备和通用磁盘对象的绑定

通用磁盘对象需要把自身注册到系统的管理链表中。这是通过 blk_register_region 函数来实现的，如代码清单 9-1 所示。

代码清单 9-1　blk_register_region

```
void blk_register_region(dev_t dev, unsigned long range, struct module *module,
          struct kobject *(*probe)(dev_t, int *, void *),
          int (*lock)(dev_t, void *), void *data){
  kobj_map(bdev_map, dev, range, module, probe, lock, data);
}
```

kobj_map 是一个熟悉的函数，在第 5 章已经分析过，作用是把设备号注册到系统的管理链表。

通过设备号获得通用磁盘对象时，需要调用 get_gendisk 函数，如代码清单 9-2 所示。

代码清单 9-2　get_gendisk

```
struct gendisk *get_gendisk(dev_t dev, int *part){
  struct kobject *kobj = kobj_lookup(bdev_map, dev, part);
  return  kobj ? to_disk(kobj) : NULL;
}
```

kobj_lookup 也是一个熟悉的函数，作用是根据设备号搜索 kobj 结构，然后通过 container 方法获得通用磁盘对象结构指针。

再次回顾 init_special_inode 函数对块设备的处理代码：

```
} else if (S_ISBLK(mode)) {
```

```
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
}
```

块设备的特殊 inode 的操作函数结构被设置为 def_blk_fops，而 inode 自身也保存了设备号，共同的设备号把块设备和通用磁盘对象关联了起来。

9.1.3 块设备的队列和队列处理函数

通用磁盘对象包含一个队列 queue，块设备的队列其实是借用这个队列。队列的结构定义如代码清单 9-3 所示。

代码清单 9-3 request_queue

```
struct request_queue
{
  /*
   * Together with queue_head for cacheline sharing
   */
  struct list_headqueue_head;

  elevator_t              *elevator;
  request_fn_proc         *request_fn;
  merge_request_fn        *back_merge_fn;
  merge_request_fn        *front_merge_fn;
  merge_requests_fn       *merge_requests_fn;
  make_request_fn         *make_request_fn;
  prep_rq_fn              *prep_rq_fn;
  unplug_fn               *unplug_fn;
  merge_bvec_fn           *merge_bvec_fn;
  activity_fn             *activity_fn;
  issue_flush_fn          *issue_flush_fn;
  prepare_flush_fn        *prepare_flush_fn;
  softirq_done_fn         *softirq_done_fn;
}
```

这个结构中最重要的有两点，一是结构中封装了一个 elevator_t 指针，二是队列中包含了众多的队列处理函数指针。

块设备一般具有连续读写快、随机读写慢的特征（硬盘这种机械装置的物理特征）。所有在块设备队列中排队的读写请求，需要经过 elevator_t 进行次序调整，然后才真正由块设备执行读写。elevator_t 结构提供了一种框架，这个框架提供不同的调度算法，后文将分析块设备的调度算法。

在内核的 I/O 处理流程中，需要调用队列中的处理函数。比如 make_request_fn 用来将 I/O 请求插入到队列，而 request_fn_proc 则用来从队列中获得一个 I/O 请求。当完成 I/O 时，调用软中断处理函数 softirq_done_fn 处理。入队列的次序和出队列的次序可能不同，这是 I/O 调度算法提供的功能。

9.2 块设备创建的过程分析

从内核中，选择位于目录drivers\block的nbd驱动作为一个简单的块设备例子，根据代码前部的说明，这个驱动是为了在网络环境中使用块设备而提供的。

这个例子可以帮助我们快速理解块设备的整体架构，了解队列、通用磁盘对象和块设备这种架构的使用方式。

9.2.1 nbd驱动的初始化

首先从nbd驱动的初始化函数开始分析，这个初始化函数是nbd_init，如代码清单9-4所示。

代码清单9-4 nbd_init (nbd.c)

```
static int __init nbd_init(void)
{
  int err = -ENOMEM;
  int i;

  BUILD_BUG_ON(sizeof(struct nbd_request) != 28);
  /* 判断nbd设备数目不能超过最大许可数目 */
  if (nbds_max > MAX_NBD) {
       printk(KERN_CRIT "nbd: cannot allocate more than %u nbds;
               %u requested.\n", MAX_NBD, nbds_max);
       return -EINVAL;
  }
  for (i = 0; i < nbds_max; i++) {
       struct gendisk *disk = alloc_disk(1);
       if (!disk)
            goto out;
       nbd_dev[i].disk = disk;

     /* 注册nbd自身的request_fn函数指针 */
       disk->queue = blk_init_queue(do_nbd_request, &nbd_lock);
       if (!disk->queue) {
            put_disk(disk);
            goto out;
       }
  }
  /* 注册块设备 */
  if (register_blkdev(NBD_MAJOR, "nbd")) {
       err = -EIO;
       goto out;
  }
```

nbd_init函数第一部分首先申请足够的通用磁盘对象，然后为每个通用磁盘对象注册它的处理函数，这是通过blk_init_queue函数实现的。

nbd_init函数第二部分设置通用磁盘对象的参数。

```
for (i = 0; i < nbds_max; i++) {
     struct gendisk *disk = nbd_dev[i].disk;
     nbd_dev[i].file = NULL;
     nbd_dev[i].magic = LO_MAGIC;
     nbd_dev[i].flags = 0;
     spin_lock_init(&nbd_dev[i].queue_lock);
     INIT_LIST_HEAD(&nbd_dev[i].queue_head);
     mutex_init(&nbd_dev[i].tx_lock);
     init_waitqueue_head(&nbd_dev[i].active_wq);
     nbd_dev[i].blksize = 1024;
     nbd_dev[i].bytesize = 0x7ffffc00ULL << 10; /* 2TB */
     /* 设置通用磁盘对象的主从设备号 */
     disk->major = NBD_MAJOR;
     disk->first_minor = i;
     /* 设置 nbd 设备的 I/O control 函数 */
     disk->fops = &nbd_fops;
     disk->private_data = &nbd_dev[i];
     disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
     sprintf(disk->disk_name, "nbd%d", i);
     set_capacity(disk, 0x7ffffc00ULL << 1); /* 2 TB */
     add_disk(disk);
  }
  return 0;
```

块设备的读写不是通过独立的读写函数实现的，而是通过队列处理函数来完成的。通用磁盘对象的操作函数结构体 nbd_fops 并不需要设置读写函数，所以只提供了 I/O control 函数。

设置通用磁盘对象的名字和设备容量之后，调用 add_disk 把磁盘对象注册到系统。

9.2.2 为通用磁盘对象创建队列成员

nbd_init 调用了 blk_init_queue 函数，目的是为通用磁盘对象创建 queue 成员，并设置队列的处理函数，如代码清单 9-5 所示。

代码清单 9-5 blk_init_queue(ll_rw_blk.c)

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
{
  return blk_init_queue_node(rfn, lock, -1);
}

request_queue_t *
blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock, int node_id)
{
  /* 创建一个队列结构 */
  request_queue_t *q = blk_alloc_queue_node(GFP_KERNEL, node_id);
  ....../* 省略部分代码 */
  /* 注册 request_fn 函数 */
  q->request_fn          = rfn;
  q->back_merge_fn       = ll_back_merge_fn;
```

```
    q->front_merge_fn          = ll_front_merge_fn;
    q->merge_requests_fn       = ll_merge_requests_fn;
    q->prep_rq_fn              = NULL;
    q->unplug_fn               = generic_unplug_device;
    q->queue_flags             = (1 << QUEUE_FLAG_CLUSTER);
    q->queue_lock              = lock;

    blk_queue_segment_boundary(q, 0xffffffff);
    /* 设置块设备入队列函数 */
    blk_queue_make_request(q, __make_request);
    /* 设置最大段的尺寸 */
    blk_queue_max_segment_size(q, MAX_SEGMENT_SIZE);
    /* 硬件能处理的最大段数目 */
    blk_queue_max_hw_segments(q, MAX_HW_SEGMENTS);
    blk_queue_max_phys_segments(q, MAX_PHYS_SEGMENTS);
    /*
     * all done
     */
     /* 申请一个elevator结构 */
    if (!elevator_init(q, NULL)) {
         blk_queue_congestion_threshold(q);
         return q;
    }

    blk_put_queue(q);
    return NULL;
}
```

blk_init_queue 函数设置了队列的入队列函数 make_request_fn 和出队列函数 request_fn，还有众多 I/O 请求合并函数以及队列闭塞函数等。为队列设置的众多参数中很多都和具体物理设备有关，这些参数的应用将在内核的 I/O 处理流程中体现。最后调用 elevator_init 初始化一个 elevator 对象，并设置默认的 I/O 调度算法。

9.2.3 将通用磁盘对象加入系统

现在返回 nbd_init 函数，分析 add_disk 函数如何把通用磁盘对象加入系统，如代码清单 9-6 所示。

代码清单 9-6 add_disk (genhd.c)

```
void add_disk(struct gendisk *disk)
{
  disk->flags |= GENHD_FL_UP;
  blk_register_region(MKDEV(disk->major, disk->first_minor),
       disk->minors, NULL, exact_match, exact_lock, disk);
  register_disk(disk);
  blk_register_queue(disk);
}
```

这个函数由三个子函数组成。blk_register_region 调用的就是 kobj_map，把通用磁盘对象的设备号加入系统的块设备链表。后续查找通用磁盘对象通过函数 kobj_lookup 执行。register_disk 创建了一个块设备对象，并完成块设备和通用磁盘对象的绑定。这个过程在块设备打开时候还会执行一次，这在后文分析。

9.3 块设备文件系统

Linux 通过块设备文件系统来管理块设备，这听起来似乎不可理解，但思索 Linux 文件系统的架构，文件系统可以看做一个对象，对象中包含超级块、inode 和文件等结构，以及它们的处理函数。而块设备是文件系统的一个文件（通过 mknod 命令生成），和字符设备 input 的架构类似，同样可以把块设备作为一个框架，注册不同的处理函数。

用面向对象的思想比较容易理解内核的这种设计思路。块设备文件系统是一种对象，这种对象提供的方法和数据可以被另外的文件系统对象使用。通读内核代码，面向对象的设计思路使用的非常普遍，I/O 调度算法和电梯 elevator 结构都可以看做封装的对象。

9.3.1 块设备文件系统的初始化

对块设备文件系统的分析需要从初始化函数 bdev_cache_init 开始，如代码清单 9-7 所示。

代码清单 9-7 bdev_cache_init

```
void __init bdev_cache_init(void)
{
  int err;
  bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode),
            0, (SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|
            SLAB_MEM_SPREAD|SLAB_PANIC),
            init_once, NULL);
  err = register_filesystem(&bd_type);
  if (err)
      panic("Cannot register bdev pseudo-fs");
      bd_mnt = kern_mount(&bd_type);
      err = PTR_ERR(bd_mnt);
      if (IS_ERR(bd_mnt))
          panic("Cannot create bdev pseudo-fs");
  blockdev_superblock = bd_mnt->mnt_sb; /* For writeback */
}
```

这个初始化函数首先把块设备文件系统注册到内核，然后调用 kern_mount 创建文件系统必要的数据结构。第 2 章已经分析过类似的文件系统初始化代码，此处不赘述。

9.3.2 块设备文件系统的设计思路

根据掌握的文件系统知识，我们从两个方面理解文件系统。

- 第一个方面是文件系统超级块对象提供的操作函数。
- 第二个方面是文件系统为 inode 结构和文件对象提供的操作函数。

首先分析块设备文件系统超级块提供的操作函数 bdev_sops，如下所示：

```
static struct super_operations bdev_sops = {
   .statfs = simple_statfs,
  .alloc_inode = bdev_alloc_inode,
  .destroy_inode = bdev_destroy_inode,
  .drop_inode = generic_delete_inode,
  .clear_inode = bdev_clear_inode,
};
```

这个结构里面，只有 bdev_alloc_inode 函数和 bdev_destroy_inode 函数是块设备文件系统提供的独特函数，其他都是借用系统的通用处理函数。函数 bdev_destroy_inode 非常简单，读者自行分析即可。

通过分析 bdev_alloc_inode 函数即可了解块设备文件系统的主要设计思路，如代码清单 9-8 所示。

代码清单 9-8 bdev_alloc_inode（block_dev.c）

```
static struct inode *bdev_alloc_inode(struct super_block *sb)
{
  struct bdev_inode *ei = kmem_cache_alloc(bdev_cachep, SLAB_KERNEL);
  if (!ei)
      return NULL;
  return &ei->vfs_inode;
}
```

块设备文件系统提供了一个独特的结构 bdev_inode。这个结构封装了一个块设备结构和一个 inode 结构。inode 结构适应文件系统统一的接口需要，而块设备结构则提供了块设备需要的功能，这就是块设备文件系统架构设计的巧妙之处。

文件系统的第二个重要方面是 inode 结构和文件对象操作函数。块设备文件系统不需要创建目录和文件，因此没有提供 inode 的操作函数。文件对象的操作函数中，读写函数使用了内核提供的通用读写函数（将在第 10 章分析），因此本节重点关注块设备的打开流程。

9.4 块设备的打开流程

通常打开块设备有两种方式，一种是直接打开块设备，也称为裸设备使用方式。通过这种方式打开块设备，实际上是调用了块设备文件系统提供的 blkdev_open 函数。如下面代码所示：

```
fopen(/dev/sda)
```

另外一种方式是通过文件系统间接使用块设备。块设备文件系统提供了 open_bdev_excl 函数。当文件系统建立在块设备之上时，需要调用这个函数来绑定块设备和文件系统，在文件系统的超级块结构中保存所在块设备的信息。

从内核代码的角度来看，这两种打开方式实际上类似。因此本节以裸设备使用方式为例进行分析。

打开一个块设备文件，最终是调用 init_special_inode 函数为块设备文件提供的默认处理函数封装结构 def_blk_fops，这个结构也就是块设备文件系统提供的文件操作函数结构。打开一个块设备，实际就调用了这个结构里面提供的 blkdev_open 函数，如代码清单 9-9 所示。

代码清单 9-9 blkdev_open（block_dev.c）

```
static int blkdev_open(struct inode * inode, struct file * filp)
{
  struct block_device *bdev;
  int res;

  filp->f_flags |= O_LARGEFILE;
  /* 获得块设备对象 */
  bdev = bd_acquire(inode);

  res = do_open(bdev, filp, BD_MUTEX_NORMAL);
  if (res)
      return res;

  if (!(filp->f_flags & O_EXCL) )
      return 0;
 /* 设置块设备和文件的参数 */
  if (!(res = bd_claim(bdev, filp)))
      return 0;

  blkdev_put(bdev);
  return res;
}
```

函数 blkdev_open 首先通过 bd_acquire 获得块设备对象，然后调用 do_open 执行块设备的打开过程。

9.4.1 获取块设备对象

首先分析 bd_acquire 函数，它的作用是从块设备文件系统获得块设备对象。本节分两部分介绍 bd_acquire 函数。

1. 块设备对象已经存在

bd_acquire 函数第一部分，判断块设备对象是否存在。如果已经存在，只增加 bd_inode 的引用计数，然后返回，如代码清单 9-10 所示。

代码清单 9-10 bd_acquire (block_dev.c)

```
static struct block_device *bd_acquire(struct inode *inode)
{
  struct block_device *bdev;

  spin_lock(&bdev_lock);
  bdev = inode->i_bdev;
  /* 如果已经有块设备对象，则只增加引用计数，然后返回 */
  if (bdev) {
      atomic_inc(&bdev->bd_inode->i_count);
      spin_unlock(&bdev_lock);
      return bdev;
  }
```

需要指出的是，操作系统启动时要扫描硬盘，对扫描到的硬盘要调用 add_disk 函数注册。注册的过程执行了块设备的打开过程，创建块设备文件系统的数据结构，如 inode 结构和块设备对象。因此当用户调用 fopen 打开块设备的时候，将直接获得已经打开的块设备对象。

2. 申请块设备对象

bd_acquire 函数第二部分，通过 bdget 函数申请块设备对象。如果成功，则 inode 的 i_mapping 设置为块设备关联 inode 的 i_mapping。i_mapping 成员是块设备读写功能的一个关键成员，后文将继续分析这个成员。

```
/* 申请块设备对象 */
bdev = bdget(inode->i_rdev);
if (bdev) {
     spin_lock(&bdev_lock);
     if (!inode->i_bdev) {
         /* 增加引用计数 */
         atomic_inc(&bdev->bd_inode->i_count);
         inode->i_bdev = bdev;
         inode->i_mapping = bdev->bd_inode->i_mapping;
         list_add(&inode->i_devices, &bdev->bd_inodes);
     }
```

bdget 函数可以分为两部分，如代码清单 9-11 所示。第一部分通过块设备文件系统分配一个 bdev_inode 对象，第二部分为这个对象设置一系列的参数，包括设备号、块设备指针等。

代码清单 9-11 bdget (block_dev.c)

```
struct block_device *bdget(dev_t dev)
{
  struct block_device *bdev;
  struct inode *inode;
  /* 创建一个 bdev_inode 结构 */
  inode = iget5_locked(bd_mnt->mnt_sb, hash(dev),
```

```
            bdev_test, bdev_set, &dev);

  if (!inode)
      return NULL;
 /* 返回 bdev_inode 结构包含的块设备结构 */
  bdev = &BDEV_I(inode)->bdev;

  if (inode->i_state & I_NEW) {
      bdev->bd_contains = NULL;
      bdev->bd_inode = inode;
      bdev->bd_block_size = (1 << inode->i_blkbits);
      bdev->bd_part_count = 0;
      bdev->bd_invalidated = 0;
      inode->i_mode = S_IFBLK;
      inode->i_rdev = dev;
      inode->i_bdev = bdev;
      inode->i_data.a_ops = &def_blk_aops;
      mapping_set_gfp_mask(&inode->i_data, GFP_USER);
      /*backing_dev_info 主要为文件的预读服务 */
      inode->i_data.backing_dev_info = &default_backing_dev_info;
      spin_lock(&bdev_lock);
      list_add(&bdev->bd_list, &all_bdevs);
      spin_unlock(&bdev_lock);
      unlock_new_inode(inode);
  }
  return bdev;
}
```

inode 结构成员 i_data 对象包含的处理函数设置为块设备文件系统提供的默认函数结构 def_blk_aops，这个结构包含的函数指针是对磁盘进行读写的底层处理函数，在后续章节中将继续分析。

分配 bdev_inode 对象通过 iget5_locked 实现，如代码清单 9-12 所示。

代码清单 9-12 iget5_locked (inode.c)

```
struct inode *iget5_locked(struct super_block *sb, unsigned long hashval,
      int (*test)(struct inode *, void *),
      int (*set)(struct inode *, void *), void *data)
{
  struct hlist_head *head = inode_hashtable + hash(sb, hashval);
  struct inode *inode;
  /* 搜索 hash 链表，寻找是否已经存在一个 inode*/
  inode = ifind(sb, head, test, data, 1);
  if (inode)
      return inode;
  return get_new_inode(sb, head, test, set, data);
}

static struct inode * get_new_inode(struct super_block *sb,
```

```
struct hlist_head *head, int (*test)(struct inode *, void *),
int (*set)(struct inode *, void *), void *data)
{
  struct inode * inode;
  /* 申请一个 inode 结构 */
  inode = alloc_inode(sb);
  if (inode) {
      struct inode * old;

      spin_lock(&inode_lock);
      /* We released the lock, so.. */
      old = find_inode(sb, head, test, data);
      if (!old) {
      ....../* 省略部分代码 */
            return inode;
      }
      /* 其他任务已经创建了 inode*/
      __iget(old);
      spin_unlock(&inode_lock);
      destroy_inode(inode);
      inode = old;
      wait_on_inode(inode);
  }
  return inode;
```

函数 iget5_locked 首先搜索 inode 的 hash 链表，如果不能找到同设备号的 inode，则创建一个新 inode 结构。创建完毕，仍然要调用 find_inode 再搜索一遍，这是因为在创建 inode 的过程中，可能有其他任务抢先创建了，这种情况下，要释放刚创建的 inode。

9.4.2 执行块设备的打开流程

现在返回 blkdev_open 函数，通过 bd_acquire 获得块设备对象后，do_open 执行块设备的打开流程。函数 do_open 分三部分介绍。

1. 获得和块设备绑定的通用磁盘对象

第一部分调用 get_gendisk 获得和块设备绑定的通用磁盘对象。

```
do_open(struct block_device *bdev, struct file *file, unsigned int subclass)
{
  struct module *owner = NULL;
  struct gendisk *disk;
  int ret = -ENXIO;
  int part;
  /* 为文件的 f_mapping 赋值。f_mapping 主要为文件的读写过程服务 */
  file->f_mapping = bdev->bd_inode->i_mapping;
  lock_kernel();
  disk = get_gendisk(bdev->bd_dev, &part);
  if (!disk) {
  /* 如果通用磁盘对象不存在，返回失败 */
```

```
        unlock_kernel();
        bdput(bdev);
        return ret;
    }
    owner = disk->fops->owner;
```

get_gendisk 函数 part 参数的目的是获得通用磁盘对象的分区信息。众所周知，硬盘可以存在多个分区，分区的主设备号相同，而次设备号不同。硬盘自身存在和它对应的块设备对象，每个分区也都有各自的块设备对象。part 为 0，说明使用的是硬盘自身的块设备对象，如果 part 不为 0，说明使用的是分区的块设备对象。

2. 块设备未被打开的处理

函数 do_open 第二部分是处理块设备未被打开，且块设备是硬盘设备自身的情况。

这种情况首先执行磁盘本身提供的 open 函数，然后设置块设备的容量信息以及后备设备信息（backing_dev_info），后备设备信息主要为设备的预读算法服务。最后如果需要扫描分区，应调用 rescan_partitions 扫描硬盘的所有分区。

```
if (!bdev->bd_openers) {/* 如果块设备未被打开 */
    bdev->bd_disk = disk;
    bdev->bd_contains = bdev;
    if (!part) {
        struct backing_dev_info *bdi;
        if (disk->fops->open) {
            ret = disk->fops->open(bdev->bd_inode, file);
            if (ret)
                goto out_first;
        }
        if (!bdev->bd_openers) {
            bd_set_size(bdev,(loff_t)get_capacity(disk)<<9);
            bdi = blk_get_backing_dev_info(bdev);
            if (bdi == NULL)
                bdi = &default_backing_dev_info;
            bdev->bd_inode->i_data.backing_dev_info = bdi;
        }
        if (bdev->bd_invalidated)
            rescan_partitions(disk, bdev);
```

3. 块设备是硬盘分区的处理

函数 do_open 第三部分处理块设备是硬盘分区的情况。

这种情况首先要获得硬盘自身块设备对象。这一步通过调用输入参数为 0 的 bdget_disk 函数实现，参数为 0 说明要求的对象索引值为 0，正是整个硬盘对应的块设备对象。

```
} else {
    struct hd_struct *p;
    struct block_device *whole;
    /* 获得磁盘索引为 0 的块设备，也就是主盘的块设备对象 */
    whole = bdget_disk(disk, 0);
    ret = -ENOMEM;
```

```
        if (!whole)
            goto out_first;
        ret = blkdev_get_whole(whole, file->f_mode, file->f_flags);
        if (ret)
            goto out_first;
        /* 设置块设备的容器 bd_contains 是整个盘，而不是自身 */
        bdev->bd_contains = whole;
        mutex_lock_nested(&whole->bd_mutex, BD_MUTEX_WHOLE);
        whole->bd_part_count++;
        p = disk->part[part - 1];
        bdev->bd_inode->i_data.backing_dev_info =
                whole->bd_inode->i_data.backing_dev_info;
        if (!(disk->flags & GENHD_FL_UP) || !p || !p->nr_sects) {
            whole->bd_part_count--;
            mutex_unlock(&whole->bd_mutex);
            ret = -ENXIO;
            goto out_first;
        }
        kobject_get(&p->kobj);
        bdev->bd_part = p;
        bd_set_size(bdev, (loff_t) p->nr_sects << 9);
        mutex_unlock(&whole->bd_mutex);
    }
```

每个分区的信息（比如分区的起始扇区地址和容量）都保存在通用磁盘对象的 part 成员里面，因此块设备的容量要根据分区的信息来设置。

第三部分整个分支执行成功的话，硬盘自身的块设备对象的分区数目要加 1。

9.5 本章小结

块设备和字符设备有明显不同，主要体现在块设备的队列，以及块设备和通用磁盘对象的关系。这种设计是基于磁盘的物理特性，为使用块设备提供了方便，不需要程序员再做额外的工作。

内核有多种块设备使用了这种架构，比如 CD、磁带、MTD 设备等，建议读者以这些设备为例子，分析它们的具体实现代码。

第 10 章
文件系统读写

Linux 系统内核为文件设置了一个缓存，对文件读写的数据内容都缓存在这里。这个缓存称为 page cache（页缓存）。

10.1 page cache 机制

page cache 是 Linux 操作系统的一个特色，其中存储的数据在 I/O 完成后并不回收，而是一直保留在内存中，除非内存紧张，才开始回收占用的内存。

10.1.1 buffer I/O 和 direct I/O

使用 page cache 的 I/O 操作称为 buffer I/O，默认情况下，内核都使用 buffer I/O；但有的应用不希望使用内核缓存，而是由应用提供内存，这种由应用提供内存的 I/O 称为 direct I/O，它的特点是不使用系统提供的 page cache。

Linux 应用编程接口提供了文件的读写接口，就是 read 和 write 接口。read 和 write 接口是同步 I/O 接口，调用这两个函数的进程会被阻塞，直到读写过程完成，才返回应用程序。和同步 I/O 接口对应的是异步 I/O 接口。异步 I/O 接口不会阻塞进程，而是立即返回。异步接口需要提供机制判断 I/O 是否完成。

Linux 系统的 buffer I/O 由于要填充 page cache，必须等读 I/O 完成才能返回，所以 buffer I/O 本身在内核中就会阻塞。所以 Linux 的异步 I/O 必须是 direct I/O，才能不阻塞进程立即返回。

10.1.2 buffer head 和块缓存

page cache 顾名思义，是以页面为单位组织的。Linux 内核对内存的管理以页面为单位，对文件缓存的管理也是以页面为单位。如果一个文件大小为 16KB，它正好可以用 4 个 4KB 的页面来缓存。因为内存有可能需要交换到硬盘上，而对硬盘文件的访问也可以通过 mmap 方式像访问内存一样进行访问。这两个管理单位的统一，减少了内核程序转换的麻烦。

硬盘这种物理介质以扇区为最小访问单位。通常一个扇区为 512 字节，对硬盘的读写最小单位是 512 字节，而文件系统是以块的方式来组织文件，文件块一般为 2 扇区、4 扇区，或者 8 扇区的格式。文件系统这种组织方式，要求提供一种块缓存机制来暂存文件的内容。所以内核提供了 buffer head 管理结构来管理块缓存。

buffer head 本身并没有保存文件内容，文件内容实际上还是在 page cache 中，buffer head 是个管理结构，它只是标识文件块的序号以及文件块缓存的地址。buffer head 同时提供对底层硬件设备（块设备）的映射。代码清单 10-1 给出了 buffer head 的结构定义。

代码清单 10-1 buffer_head 结构定义

```
struct buffer_head {
  unsigned long b_state;            /* buffer state bitmap (see above) */
  struct buffer_head *b_this_page;  /* circular list of page's buffers */
  struct page *b_page;              /* the page this bh is mapped to */

  sector_t b_blocknr;               /* start block number */
  size_t b_size;                    /* size of mapping */
  char *b_data;                     /* pointer to data within the page */

  struct block_device *b_bdev;
  bh_end_io_t *b_end_io;            /* I/O completion */
  void *b_private;                  /* reserved for b_end_io */
  struct list_head b_assoc_buffers; /* associated with another mapping */
  atomic_t b_count;                 /* users using this buffer_head */
};
```

解释一下 buffer head 数据结构的重要成员。

- b_this_page：buffer head 单向链表，指向下一个 buffer head 结构。
- b_page：指向数据所在的页面。
- b_blocknr：buffer head 的起始块号。这个块号是以整个硬盘为空间编址的，所以可以转换为硬盘的物理扇区地址。
- b_data：指向数据的地址。
- b_bdev：文件系统绑定的块设备。
- b_end_io：回调函数。I/O 处理完毕后调用这个函数。

b_blocknr 是以整个硬盘为空间编址，这个信息只有文件系统可以知道。在第 9 章分析了文件系统打开块设备的过程，文件系统的超级块对象保存了块设备指针，通过块设备指针可以获得硬盘的容量信息和硬盘分区的信息，同时文件的数据空间是由文件系统分配的，因此文件系统知道硬盘上的数据分布，可以提供以整个硬盘为编址空间的块号。硬盘文件系统一般提供 get_block 调用将文件的位置翻译为硬盘的块号信息。

10.1.3 page cache 的管理

通过数据结构 address_space 管理 page cache。这个数据结构提供了一个 radix tree 成员，

文件内容的缓存页保存在这个 radix tree 里面。

对 page cache 而言，最重要的调用有两个，一是插入页面到 page cache，另一个是从 page cache 搜索页面。前者通过 add_to_page_cache 来实现，如代码清单 10-2 所示。

代码清单 10-2 add_to_page_cache（filemap.c）

```
int add_to_page_cache(struct page *page, struct address_space *mapping,
      pgoff_t offset, gfp_t gfp_mask)
{
  int error = radix_tree_preload(gfp_mask & ~__GFP_HIGHMEM);

  if (error == 0) {
      write_lock_irq(&mapping->tree_lock);
      error = radix_tree_insert(&mapping->page_tree, offset, page);
      if (!error) {
          page_cache_get(page);
          SetPageLocked(page);
          page->mapping = mapping;
          page->index = offset;
          mapping->nrpages++;
          __inc_zone_page_state(page, NR_FILE_PAGES);
      }
      write_unlock_irq(&mapping->tree_lock);
      radix_tree_preload_end();
  }
  return error;
}
```

这个函数逻辑很清晰，首先是创建 radix tree 根节点，然后把页面加入到 radix tree。加入成功后，设置页面的 index。

从 page cache 搜索一个页面通过 find_get_page 实现。这个函数实现很简单，不再分析。

10.1.4 page cache 的状态

页面有多种状态，由于内存管理的页和 page cache 的页是同一个结构，所以页面的状态其实也包含了 page cache 页面需要的状态。解释几个 page cache 中比较重要的状态。

- PG_uptodate：页包含最新有效的数据。当读与该页对应的文件内容时，可以直接把页的内容复制给调用者。
- PG_dirty：页包含脏数据，需要写入到硬盘。
- PG_private：页私有属性。在 page cache 中，设置私有属性意味着为页创建了块缓存结构（buffer head），同时页面数据结构的 private 成员指向块缓存结构的头指针。
- PG_mappedtodisk：页已经映射到硬盘。页映射到硬盘意味着这个页包含的所有块都已经映射到了硬盘。

块缓存也有多种状态，解释其中比较重要的几种状态。

- BH_Mapped：块已经映射到硬盘。这个块通过调用文件系统的 get_block 已经获得了

底层块设备的指针和以整个硬盘为编址空间的文件块号。

- BH_Uptodate：块缓存包含最新有效的数据。
- BH_Dirty：块缓存包含脏数据，需要写入硬盘。

10.2 文件预读

对于文件读请求，Linux 内核提供了预读策略，比要求读的长度要多读一些，存储在 page cache 里，后续读如果是顺序的，马上可以利用 page cache 的数据返回，不必再次读硬盘。对于硬盘这种慢速设备而言，利用缓存数据可以大大提升 I/O 的效率。

内核提供了默认的预读参数，如代码清单 10-3 所示。

代码清单 10-3 default_backing_dev_info

```
struct backing_dev_info default_backing_dev_info = {
  .ra_pages         = (VM_MAX_READAHEAD * 1024) / PAGE_CACHE_SIZE,
  .state            = 0,
  .capabilities     = BDI_CAP_MAP_COPY,
  .unplug_io_fn     = default_unplug_io_fn,
```

VM_MAX_READAHEAD 默认设置为 128KB，也就是默认预读页面是 32 个 4KB 页面。

Linux 内核会根据文件读是否顺序启动预读参数和设置预读窗口，对于连续的顺序读，会尽量多读一些内容填充 page cache。


提示

这部分内容在 readahead.c 文件里面，文件本身不大，也比较孤立，不涉及太多关联的知识点，读者可以作为一个例子，分析一下预读代码。

10.3 文件锁

如果一个进程对其他进程正在读取的文件进行写操作，虽然每次读写调用都是原子的，但是读写调用之间并没有同步，因此可能导致读进程读到被破坏或者不完整的数据。为了避免这种问题出现，必须有某种机制避免多进程并发访问的冲突问题。Linux 提供的机制就是文件锁。根据实现机制的不同，文件锁可分为建议锁和强制锁两种类型。

- **建议锁**：由应用层实现，内核只为用户提供程序接口，并不参与锁的控制和协调，也不对读写操作做内部检查和强制保护。如果有进程不检查文件是否有建议锁就写入数据，内核不加以阻拦。所以建议锁要求进程都遵守规则。建议锁可以对整个文件加锁，也可以只对文件的一部分加锁。
- **强制锁**：由内核强制实施。只要有进程调用读写操作，内核都会检查与存在的锁是否冲突，如果冲突，内核就会加以阻拦。

根据访问方式的不同，文件锁又分为读锁和写锁。

- **读锁**：允许多个进程同时进行读操作，又称**共享锁**。文件加了读锁就不能再设置写锁，但允许其他进程在同一区域再设置读锁。
- **写锁**：主要目的是隔离文件，使所写内容不被其他进程的读写干扰，以保证数据的完整性。写锁一旦加上，只有上锁的人可以操作，其他进程无论读还是写都只能等待写锁释放后才能执行，故写锁又称**互斥锁**。

如果一个文件已经被加上了读锁，其他进程再对这个文件进行写操作就会被内核阻止。如果一个文件已经被加上了写锁，其他进程再对这个文件进行读取或者写操作就会被内核阻止。

10.4 文件读过程代码分析

为了便于理解文件的读写操作，图 10-1 给出一个例子文件的内容分布图。

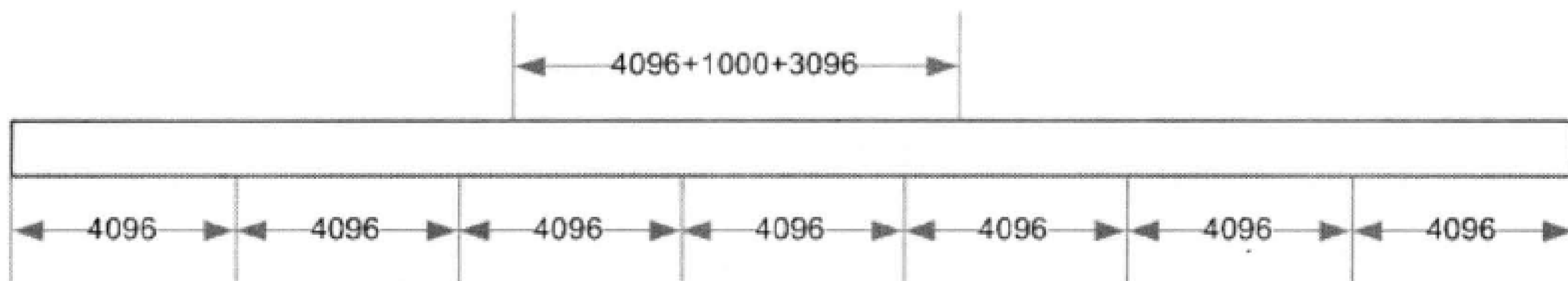


图 10-1 文件内容分布图

文件总长度 4096 字节 ×7=28672 字节。从 4096 字节 ×2+1000 字节的位置开始读，读 4096 字节 +1000 字节 +3096 字节。同时，设定文件开始在硬盘的第 10000 个扇区，前面我们已经知道硬盘扇区是 512 字节，文件的起始位置就是 10000×512 字节。

内核处理读文件从 sys_read 函数开始，我们就从这个函数开始读过程分析。sys_read 函数的实现代码如代码清单 10-4 所示。

代码清单 10-4 sys_read (read_write.c)

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
{
  struct file *file;
  ssize_t ret = -EBADF;
  int fput_needed;
  file = fget_light(fd, &fput_needed);
  if (file) {
      /* 获取文件的当前位置 */
      loff_t pos = file_pos_read(file);
      ret = vfs_read(file, buf, count, &pos);
      file_pos_write(file, pos);
      fput_light(file, fput_needed);
  }
```

```
  return ret;
}
```

sys_read 函数首先根据文件 ID 获得文件结构的指针。每个进程都有一个 files_struct 结构指针，它保存了进程所有打开的文件，因此以文件 ID 为索引，就可以获得文件结构指针。其次取得文件的当前位置，这个参数是文件系统内部保存，每次执行读函数调用，都要记录读操作的最后位置，以备下次操作使用。

最后调用 vfs_read 函数执行文件读，读完之后，要把更新的文件当前位置写入文件指针。vfs_read 函数的实现代码如代码清单 10-5 所示。

代码清单 10-5 vfs_read（read_write.c）

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  ....../* 省略部分代码 */
  /* 校验文件的锁 */
      ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
      count = ret;
      ret = security_file_permission (file, MAY_READ);
      if (!ret) {
          if (file->f_op->read)
              ret = file->f_op->read(file, buf, count, pos);
          else
              ret = do_sync_read(file, buf, count, pos);
```

vfs_read 函数首先检查文件读写锁的权限。如果文件不支持强制锁，这个检查直接通过，如果支持强制锁，就要按前一节的描述检查锁是否冲突。

如果文件定义了 read 函数，调用文件自身的读函数，否则的话，系统提供了一个函数 do_sync_read 作为读函数。文件系统的函数是如何注册到文件的 f_op 指针？这是文件初始化期间生成 inode 结构时赋予的。数据文件、目录文件或者设备文件有各自不同的读写函数，这在第 2 章已经分析过，此处不再重复。

不同文件系统定义了不同的读写函数（很多也是相同的）。为了便于分析，我们选择一个广泛使用的文件系统——ext2 文件系统作为例子，它的读写函数具有普遍的意义。

1. generic_file_read 函数

ext2 文件系统的读函数使用了 generic_file_read。从名字可以看出，这个函数是个通用函数，实际上很多文件系统都使用了这个函数，如代码清单 10-6 所示。

代码清单 10-6 vfs_read → generic_file_read（filemap.c）

```
ssize_t
generic_file_read(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
{
  struct iovec local_iov = { .iov_base = buf, .iov_len = count };
```

```
    struct kiocb kiocb;
    ssize_t ret;

    init_sync_kiocb(&kiocb, filp);
    ret = __generic_file_aio_read(&kiocb, &local_iov, 1, ppos);
    if (-EIOCBQUEUED == ret)
        ret = wait_on_sync_kiocb(&kiocb);
    return ret;
}
```

generic_file_read 函数主要解决文件同步操作和异步操作的问题，这是通过一个同步控制结构 kiocb 实现的。函数开始调用 init_sync_kiocb 初始化一个同步控制块 kiocb，然后将读操作异步提交，如果读操作返回 EIOCBQUEUED（说明读操作未完成，尚在队列中，需要等待操作完成），进程置为睡眠态，等待 kiocb 的成员 ki_users 变为 0。kiocb 结构的定义在文件 \include\aio.h 中，而它的控制逻辑主要在内核的异步 I/O 实现文件 aio.c 中。

前面的章节中分析过，真正的异步操作是很难实现的。使用 page cache 的 buffer I/O 时因为要等待读 I/O 完成才能返回，这个过程有可能阻塞进程，所以 buffer I/O 的实现过程本身就不能保证异步，等 buffer I/O 读过程返回，实际上已经完成了读操作。

2. __generic_file_aio_read 函数

generic_file_read 函数最后调用 __generic_file_aio_read 执行文件读，输入参数 iov 包含用户传入的用户态地址和希望读的字节数，如代码清单 10-7 所示。

代码清单 10-7 __generic_file_aio_read (filemap.c)

```
ssize_t __generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,
        unsigned long nr_segs, loff_t *ppos)
{
  ....../* 省略部分代码 */
  count = 0;
  /* 计算希望读文件的字节数 */
  for (seg = 0; seg < nr_segs; seg++) {
      const struct iovec *iv = &iov[seg];
      /*
       * If any segment has a negative length, or the cumulative
       * length ever wraps negative then return -EINVAL.
       */
      count += iv->iov_len;
      if (unlikely((ssize_t)(count|iv->iov_len) < 0))
          return -EINVAL;
      /* 检查用户态地址是否合法 */
      if (access_ok(VERIFY_WRITE, iv->iov_base, iv->iov_len))
          continue;
      if (seg == 0)
          return -EFAULT;
      nr_segs = seg;
      count -= iv->iov_len;      /* This segment is no good */
```

```
        break;
    }
```

__generic_file_aio_read 函数第一部分计算希望读文件的字节数，校验用户态地址是否合法。在当前场景，已经限制了 nr_segs 为 1，所以 iovec 只有一个向量。参数 count 得到了希望读文件的字节数，也就是我们前面赋予的 4096 字节 +1000 字节 +3096 字节。access_ok 是校验用户态地址是否合法，我们知道在 32 位计算机系统中，Linux 用户态的地址空间从 0 到 3G 字节（通常的情况），这个函数经常在内核用到。

__generic_file_aio_read 函数后面部分是对 direct I/O 的处理以及底层调用，如下所示。

```
retval = 0;
  if (count) {
      for (seg = 0; seg < nr_segs; seg++) {
           read_descriptor_t desc;
           /* 赋值读描述符 */
           desc.written = 0;
           desc.arg.buf = iov[seg].iov_base;
           desc.count = iov[seg].iov_len;
           if (desc.count == 0)
               continue;
           desc.error = 0;
           do_generic_file_read(filp,ppos,&desc,file_read_actor);
           retval += desc.written;
           if (desc.error) {
               retval = retval ?: desc.error;
               break;
           }
      }
  }
out:
  return retval;
}
```

__generic_file_aio_read 函数第二部分处理 direct I/O。direct I/O 只是业务逻辑，和 buffer I/O 基本流程大部分是相同的，所以本节重点在 buffer I/O，略过 direct I/O。

__generic_file_aio_read 函数第三部分使用了读描述符结构 desc。desc 的 written 成员代表从文件中读到的字节数，当调用 do_generic_file_read 函数返回后，written 就等于此时读到的字节数，而函数指针 file_read_actor 用来将数据从内核的 page cache 复制到用户提供的 buf，也就是 iov_base 地址描述的内存。

3. do_generic_file_read 函数

do_generic_file_read 函数是内核提供的通用读函数，如代码清单 10-8 所示。

代码清单 10-8 do_generic_file_read (filemap.c)

```
static inline void do_generic_file_read(struct file * filp, loff_t *ppos,
                    read_descriptor_t * desc, read_actor_t actor)
```

```
{
  do_generic_mapping_read(filp->f_mapping,
                &filp->f_ra,
                filp,
                ppos,
                desc,
                actor);
}
```

do_generic_file_read 函数封装了 do_generic_mapping_read。输入参数 f_mapping 封装了块设备的读页面和写页面函数。对于 ext2 文件系统，它在文件 inode 初始化的时候设置了读写页面函数结构 ext2_aops，打开文件的时候，设置文件的 f_mapping 等于 inode 结构提供的结构指针。

4. do_generic_mapping_read 函数

函数 do_generic_mapping_read 要计算文件读操作涉及的页面参数。这个函数非常庞杂，因此分为七个部分解释。

对于使用了 page cache 的 buffer I/O，文件的操作要落地到对 page cache 的操作。如果文件的内容已经在 page cache 里面，不需要读，直接复制内存就可以；如果 page cache 没有文件内容，则需要申请 page cache，然后从硬盘读文件内容到 page cache。为完成对 page cache 的查找，首先要把文件读的字节数计算为 page cache 使用的页面索引值，然后才便于查找。第一部分如代码清单 10-9 所示。

代码清单 10-9 do_generic_mapping_read (filemap.c)

```
872 void do_generic_mapping_read(struct address_space *mapping,
873                 struct file_ra_state *_ra,
874                 struct file *filp,
875                 loff_t *ppos,
876                 read_descriptor_t *desc,
877                 read_actor_t actor)
878 {
      ....../* 省略部分代码 */
      /* 计算读文件的第一个页面 */
892   index = *ppos >> PAGE_CACHE_SHIFT;
893   next_index = index;
894   prev_index = ra.prev_page;
895   last_index = (*ppos + desc->count + PAGE_CACHE_SIZE-1) >> PAGE_CACHE_SHIFT;
896   offset = *ppos & ~PAGE_CACHE_MASK;
897
898   isize = i_size_read(inode);
899   if (!isize)
900       goto out;
901
902   end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
903   for (;;) {
904       struct page *page;
```

```
905         unsigned long nr, ret;
906
907         /* nr is the maximum number of bytes to copy from this page */
908         nr = PAGE_CACHE_SIZE;
909         if (index >= end_index) {
910             if (index > end_index)
911                 goto out;
912             nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
913             if (nr <= offset) {
914                 goto out;
915             }
916         }
917         nr = nr - offset;
918
919         cond_resched();
920         if (index == next_index)
921             next_index = page_cache_readahead(mapping, &ra, filp,
922                     index, last_index - index);
```

第892行计算读文件的第一个页面。文件读设定从文件位置4096字节 ×2+1000字节开始读，计算得到index就是2。prev_index又是什么？它得到文件上次操作读到的最后一个页面。last_index计算读的最后一个页面，得到结果5。第892行和895行是内核常用的计算页面对齐的方法，第892行是向下对齐，而895行则是向上对齐。第896行offset计算文件位置在页面内的字节数，计算得到1000字节。第902行则是计算文件的最后一个页面索引。前面设定文件长度为7×4096字节，end_index计算得到6。注意end_index是向下对齐的。

从第903行开始循环遍历所有的页面，检查页面是否在page cache之内，如果是则从page cache直接复制到用户buf，否则需要申请页面，然后从硬盘读入页面数据，再复制到用户buf。

首先第908行nr设为4096，这是一个页面内最大可读的字节数。第909 ~ 916行则是判断是否已经读到文件末尾的情况。如果是文件最后一个页面，则要根据文件的总长度调整能读到的字节数。

第917行计算页面内可读的字节数。因为我们是从4096×2+1000字节开始读，第一个页面能读的字节就是3096字节。

第920行，因为是第一次循环，所以index等于next_index，调用page_cache_readahead进入文件预读。传入的参数指明是从第二个页面开始读，读三个页面，读完后，正常情况下，next_index得到5。

do_generic_mapping_read函数第二部分检查page cache是否存在我们需要的页面。

```
924 find_page: /* 在page cache中查找page*/
925         page = find_get_page(mapping, index);
926         if (unlikely(page == NULL)) {
927             handle_ra_miss(mapping, &ra, index);
928             goto no_cached_page;
929         }
```

```
930         if (!PageUptodate(page))
931             goto page_not_up_to_date;
```

第 924 行的 find_page 用来在 page cache 里面搜索页面。结果可以分为三种情况：

- ❑页面存在而且是最新的，只需要调用传进来的函数指针 actor，将 page cache 的内容复制到用户 buffer 即可。
- ❑页面存在，但不是最新的内容，进入 page_not_up_to_date 分支读这个页面，获得最新的内容。页面更新后，将最新内容复制到用户 buffer。
- ❑页面根本不存在，进入 no_cached_page 分支申请一个页面，然后再去读这个页面，读完成后，将最新内容复制到用户 buffer。

下面代码是对 page cache 内不同页面状态的处理。

```
932 page_ok:
            ....../* 省略部分注释 */
938         if (mapping_writably_mapped(mapping))
939             flush_dcache_page(page);
940
945         if (prev_index != index)
946             mark_page_accessed(page);
947         prev_index = index;
            ....../* 省略部分注释 */
959         ret = actor(desc, page, offset, nr);
960         offset += ret;
961         index += offset >> PAGE_CACHE_SHIFT;
962         offset &= ~PAGE_CACHE_MASK;
963
964         page_cache_release(page);
965         if (ret == nr && desc->count)
966             continue;
967         goto out;
```

do_generic_mapping_read 函数第三部分处理页面状态是最新的情况。

这种情况说明 page cache 中有要读的文件内容，调用 actor 函数将数据复制到用户态缓冲区。第 965 行判断是否已经完整复制了要读的数据，如果复制结束则退出，否则继续处理下一个页面。其次是页面在 page cache 中存在，但不是最新这种情况的处理。

```
969 page_not_up_to_date: /* 页存在，但不是最新 */
970         /* Get exclusive access to the page ... */
971         lock_page(page);
972
973         /* Did it get unhashed before we got the lock? */
974         if (!page->mapping) {
975             unlock_page(page);
976             page_cache_release(page);
977             continue;
978         }
979
980         /* Did somebody else fill it already? */
```

```
981         if (PageUptodate(page)) {
982             unlock_page(page);
983             goto page_ok;
984         }
985
```

do_generic_mapping_read 函数第四部分处理页面在 page cache 中，但是数据不是最新的情况。

此时要调用 lock_page 锁页面，这个函数可能阻塞进程，因为可能有别的进程正在读写这个页面，等其他进程读写完毕后，进程被唤醒继续执行。第 981 行第二次检查页面是否更新，因为锁页面导致进程阻塞时候，可能其他进程已经把数据读入页面了。如果页面状态未更新，继续执行进入 readpage 分支。其次处理页面中数据不存在或者虽然存在但不是最新，因此需要读页面的情况。

```
986 readpage:
987         /* Start the actual read. The read will unlock the page. */
988         error = mapping->a_ops->readpage(filp, page);
989
990         if (unlikely(error)) {
991             if (error == AOP_TRUNCATED_PAGE) {
992                 page_cache_release(page);
993                 goto find_page;
994             }
995             goto readpage_error;
996         }
997
998         if (!PageUptodate(page)) {
999             lock_page(page);
1000            if (!PageUptodate(page)) {
1001                if (page->mapping == NULL) {
1002                    /*
1003                     * invalidate_inode_pages got it
1004                     */
1005                    unlock_page(page);
1006                    page_cache_release(page);
1007                    goto find_page;
1008                }
1009                unlock_page(page);
1010                error = -EIO;
1011                shrink_readahead_size_eio(filp, &ra);
1012                goto readpage_error;
1013            }
1014            unlock_page(page);
1015        }
            ....../* 省略部分代码 */
1025        isize = i_size_read(inode);
1026        end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
1027        if (unlikely(!isize || index > end_index)) {
1028            page_cache_release(page);
```

```
1029            goto out;
1030        }
            ....../* 省略部分代码 */
1044        readpage_error:
1045        /* UHHUH! A synchronous read error occurred. Report it */
1046        desc->error = error;
1047        page_cache_release(page);
1048        goto out;
```

do_generic_mapping_read 函数第五部分处理读页面操作。

此时调用文件系统提供的 readpage 函数真正从硬盘读入一个页面的数据，因为 readpage 函数只是将读命令发送到硬盘，真正的读到数据必须等中断返回，所以第 999 行再次锁页面会导致进程睡眠（如果页面数据不是最新），直到中断返回，在中断处理函数中将页面状态改为最新并唤醒进程，此时页面状态已经是最新，从第 1025 ~ 1041 行和第 908 ~ 917 行的处理一样，设置参数值后进入 page_ok 分支开始复制数据。

上述情况有一个例外，如果中断返回后，页面状态仍不是最新，说明发生了错误，进入 readpage_error 分支释放页面后返回。第 1001 行检查 page 的 mapping 成员是否为空，如果为空，说明发生了 invalidate_inode_pages 事件，这种情况返回 find_page 分支继续处理。

```
1050 no_cached_page:
1055        if (!cached_page) {
1056            cached_page = page_cache_alloc_cold(mapping);
1057            if (!cached_page) {
1058                desc->error = -ENOMEM;
1059                goto out;
1060            }
1061        }
1062        error = add_to_page_cache_lru(cached_page, mapping,
1063                index, GFP_KERNEL);
            ....../* 省略部分代码 */
1070        page = cached_page;
1071        cached_page = NULL;
1072        goto readpage;
1073    }
```

do_generic_mapping_read 函数第六部分申请一个页面，然后将页面插入 page cache。如果成功，则进入 readpage 分支开始从硬盘读入数据。

```
1075 out:
1076    *_ra = ra;
1077
1078    *ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
1079    if (cached_page)
1080        page_cache_release(cached_page);
1081    if (filp)
1082        file_accessed(filp);
```

do_generic_mapping_read 函数第七部分更新文件的位置，修改文件的预读状态。

从硬盘读数据是通过文件系统提供的readpage函数实现的。而ext2文件系统提供的读页面函数就是ext2_readpage，如代码清单10-10所示。

代码清单 10-10 ext2_readpage（inode.c）

```
static int ext2_readpage(struct file *file, struct page *page)
{
  return mpage_readpage(page, ext2_get_block);
}
int mpage_readpage(struct page *page, get_block_t get_block)
{
  struct bio *bio = NULL;
  sector_t last_block_in_bio = 0;
  struct buffer_head map_bh;
  unsigned long first_logical_block = 0;
  /* 清除 map_bh 的映射标志 */
  clear_buffer_mapped(&map_bh);
  bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
            &map_bh, &first_logical_block, get_block);
  /* 如果 bio 有效，则发送一个读请求 bio*/
  if (bio)
      mpage_bio_submit(READ, bio);
  return 0;
}
```

ext2_readpage函数没执行任何动作，直接调用了mpage_readpage。后者是内核提供的一个通用函数，它调用do_mpage_readpage将读请求转换为一个bio结构，如果bio有效，则提交bio给底层去执行读操作。

文件系统的读写请求，最终要转换成对块设备的读写请求。这涉及几个问题。

- 文件对用户呈现了一个连续的读写接口，但是文件在真正物理设备硬盘上的存储可能并不是连续的，如果是不连续的，对文件的读写就不能用同一个I/O完成，而是需要拆分。
- 硬盘的读写最小单元是扇区，通常一个扇区是512字节，而文件的最小单元是块，一个块可以由多个扇区组成。组成块的扇区物理地址必须连续，而块之间可以不连续。
- 内核通过submit_bio来提交一个I/O给底层。同时内核又提供了一个函数submit_bh来提交块。submit_bh最终也是通过submit_bio来实现，它只是多了将块地址转换为硬盘物理扇区地址的过程。

5. do_mpage_readpage 函数

do_mpage_readpage函数必须对上面的问题做出处理，对一个页面包含的块进行检查，判断读文件的请求是否通过一个bio即可提交，还是需要拆分为多个bh，如代码清单10-11所示。

代码清单 10-11 do_mpage_readpage 函数

```
175 static struct bio *
176 do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
177         sector_t *last_block_in_bio, struct buffer_head *map_bh,
178         unsigned long *first_logical_block, get_block_t get_block)
179 {
     ....../* 省略部分代码 */
196    if (page_has_buffers(page))
197        goto confused;
198
199    block_in_file = (sector_t)page->index << (PAGE_CACHE_SHIFT - blkbits);
200    last_block = block_in_file + nr_pages * blocks_per_page;
201    last_block_in_file = (i_size_read(inode) + blocksize - 1) >> blkbits;
       /* 如果最后一个块超过文件长度，则以文件长度为准 */
202    if (last_block > last_block_in_file)
203        last_block = last_block_in_file;
204    page_block = 0;
```

do_mpage_readpage 函数第一部分首先判断当前页面是否已经创建了块缓存。

page_has_buffers 通过检查 page 结构是否设置 PG_private 标志位，如果设置了这个标志位，说明该页面已经创建了管理块缓存的 buffer head 结构。有的块缓存可能已经有了最新数据，有的块缓存可能还没有数据，这种情况转入 confused 分支，逐个检查每个块的具体情况。如果没有设置 PG_private 标志位，则要获得每个块的块号。

第 199 行计算第一个块在文件中的块号，第 200 行计算最后一个块的块号，如果最后一个块超过文件长度，则以文件长度为准。

```
     /*
207   * Map blocks using the result from the previous get_blocks call first.
208   */
209  nblocks = map_bh->b_size >> blkbits;
210  if (buffer_mapped(map_bh) && block_in_file > *first_logical_block &&
211           block_in_file < (*first_logical_block + nblocks)) {
212      unsigned map_offset = block_in_file - *first_logical_block;
213      unsigned last = nblocks - map_offset;
214
215      for (relative_block = 0; ; relative_block++) {
216           if (relative_block == last) {
217               clear_buffer_mapped(map_bh);
218               break;
219           }
220           if (page_block == blocks_per_page)
221               break;
222           blocks[page_block] = map_bh->b_blocknr + map_offset +
223                         relative_block;
224           page_block++;
225           block_in_file++;
226      }
```

```
227        bdev = map_bh->b_bdev;
228    }
```

do_mpage_readpage 函数第二部分处理页面已经映射过的情况。

函数第一部分计算的块号是文件内的逻辑块号，而要从硬盘读数据，必须根据逻辑块号获得硬盘的物理块号，这个过程就是映射。函数指针 get_block 就是文件系统提供的映射函数。

局部变量 blocks 是个数组，当前页面中每一个块的物理块号都保存在这个数组中。如果当前页面已经映射过，循环把文件块的物理块号写入 blocks 数组，直到页面内的所有块都被处理完毕。

do_mpage_readpage 函数随后部分检查每个文件块的状态，代码如下：

```
231  / * Then do more get_blocks calls until we are done with this page.
232     */
233  map_bh->b_page = page;
     /* 循环处理要读的所有块 */
234  while (page_block < blocks_per_page) {
235      map_bh->b_state = 0;
236      map_bh->b_size = 0;
237
238      if (block_in_file < last_block) {
239          map_bh->b_size = (last_block-block_in_file) << blkbits;
             /*get_block 是文件系统提供，用来获得文件块的物理块号 */
240          if (get_block(inode, block_in_file, map_bh, 0))
241              goto confused;
242          *first_logical_block = block_in_file;
243      }
244      /* 未被映射。有可能是文件中的洞 */
245      if (!buffer_mapped(map_bh)) {
246          fully_mapped = 0;
247          if (first_hole == blocks_per_page)
248              first_hole = page_block;
249          page_block++;
250          block_in_file++;
251          clear_buffer_mapped(map_bh);
252          continue;
253      }
         /* 如果这个块的内容已经被缓存而且是最新的 */
261      if (buffer_uptodate(map_bh)) {
262          map_buffer_to_page(page, map_bh, page_block);
263          goto confused;
264      }
265      /* 处理文件的洞 */
266      if (first_hole != blocks_per_page)
267          goto confused;     /* hole -> non-hole */
268
269      /* Contiguous blocks? */
270      if (page_block && blocks[page_block-1] != map_bh->b_blocknr-1)
271          goto confused;
```

```
272      nblocks = map_bh->b_size >> blkbits;
         /* 如果从文件系统一次 get_block 了多个块，则循环处理 */
273      for (relative_block = 0; ; relative_block++) {
274           if (relative_block == nblocks) {
275               clear_buffer_mapped(map_bh);
276               break;
277           } else if (page_block == blocks_per_page)
278               break;
279           blocks[page_block] = map_bh->b_blocknr+relative_block;
280           page_block++;
281           block_in_file++;
282      }
283      bdev = map_bh->b_bdev;
284  }
```

do_mpage_readpage 函数第三部分循环遍历每一个文件块，调用 get_block 获得它的物理块号。

第 245 行检查文件块如果未被映射，说明可能是文件中的空洞，则处理下一个块。

第 261 行检查 map_bh 中的内容已经是最新的，说明当前块最新的数据已经在 page cache，即使文件块的物理地址连续，也需要拆分 I/O（避免重复读硬盘），因此转入 confused 分支。

第 270 行检查文件块的物理块号和 map_bh 的物理块号不连续，说明不是顺序 I/O，需要跳转 confuse 分支。

因为一次调用 get_block 可以获得多个文件块的物理块号，所以从第 272 ~ 282 行逐个将映射获得的文件块写入 blocks 数组，写入 blocks 数组的数目不超过一个页面容纳的文件块数。变量 nblocks 保存了映射获得的文件块数目。

do_mpage_readpage 函数随后部分检查 I/O 能否合并的情况，代码如下：

```
301  /*
302   * This page will go to BIO.  Do we need to send this BIO off first?
303   */
     /* 上一个 bio 的最后块和当前 I/O 的不连续，则先发送上一个 bio*/
304  if (bio && (*last_block_in_bio != blocks[0] - 1))
305      bio = mpage_bio_submit(READ, bio);
```

do_mpage_readpage 函数的传入参数有一个 bio 指针和 bio 中最后一个块的物理块号。如果最后一个块的物理块号和 blocks 数组的物理块号连续，说明传入的 bio 可以和当前 I/O 合并，否则就必须立即把传入的 bio 进行提交，交给底层处理。

do_mpage_readpage 函数第四部分用于处理 I/O 合并，如果前一个 bio 和当前 I/O 要读的物理块号连续，两个 I/O 可以进行合并。随后部分处理 I/O 不能合并，因此需要申请新 bio 结构的情况。

do_mpage_readpage 函数第五部分申请一个新的 bio 结构。代码如下：

```
307 alloc_new:
308   if (bio == NULL) {
```

```
309         bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),
310                  min_t(int, nr_pages, bio_get_nr_vecs(bdev)),
311                  GFP_KERNEL);
312         if (bio == NULL)
313             goto confused;
314     }
315
316     length = first_hole << blkbits;
317     if (bio_add_page(bio, page, length, 0) < length) {
318         bio = mpage_bio_submit(READ, bio);
319         goto alloc_new;
320     }
321
322     if (buffer_boundary(map_bh) || (first_hole != blocks_per_page))
323         bio = mpage_bio_submit(READ, bio);
324     else
325         *last_block_in_bio = blocks[blocks_per_page - 1];
326 out:
327     return bio;
```

这个 bio 的起始扇区地址根据 blocks 数组第一个块的物理块号计算得出。如果存在文件空洞或者 map_bh 有边界标志，说明这个 bio 不能和随后的 I/O 合并，则调用 mpage_bio_submit 立即提交 I/O。否则，返回最后一个块的物理块号和 bio 地址，由系统判断何时提交 bio。

do_mpage_readpage 函数随后检查是否递交 bio 结构给下层，代码如下：

```
confused:
  if (bio)
      bio = mpage_bio_submit(READ, bio);
  if (!PageUptodate(page))
        block_read_full_page(page, get_block);
  else
      unlock_page(page);
  goto out;
}
```

do_mpage_readpage 函数第六部分是 confused 分支代码。

如果页面状态是 PG_uptodate，说明 page cache 的当前页面已经全部是最新数据，不需要从硬盘读数据，直接返回，否则就要调用 block_read_full_page 函数逐个遍历页面内的每一个文件块，检查数据是否最新。

6. block_read_full_page 函数

block_read_full_page 函数与 do_mpage_readpage 前面部分的流程类似，所以只分析最后部分，如代码清单 10-12 所示。

代码清单 10-12 block_read_full_page

```
int block_read_full_page(struct page *page, get_block_t *get_block)
{
```

```
    ....../* 省略部分代码 */
    /*
     * Stage 3: start the IO.  Check for uptodateness
     * inside the buffer lock in case another process reading
     * the underlying blockdev brought it uptodate (the sct fix).
     */
    for (i = 0; i < nr; i++) {
         bh = arr[i];
         if (buffer_uptodate(bh))
              end_buffer_async_read(bh, 1);
         else
              submit_bh(READ, bh);
    }
    return 0;
}
```

局部变量 arr 是个数组，保存了页面内的每一个文件块的物理块号。如果文件块号的状态为 BH_Uptodate，说明块的数据已经是最新，不需要从硬盘读数据，否则调用 submit_bh 把块提交给底层。

10.5 读过程返回

文件系统通过 mpage_bio_submit 提交一个 I/O。这个 I/O 什么时候返回？返回通过什么机制通知上层？这涉及内核 I/O 过程的阻塞点设计。

本章开始部分分析了同步 I/O 和异步 I/O，Linux 的同步 I/O 调用了文件系统提供的 read 函数，这个 read 函数最终要调用 do_generic_mapping_read 把文件内容按照页面读入 page cache。这个过程实际上是阻塞的，读页面的过程中两次调用了 lock_page 函数，而这个函数使用了 __wait_on_bit_lock 可能让进程进入睡眠状态。如果进程进入睡眠状态，谁来唤醒它？这就要靠 mpage_bio_submit 注册的回调函数了。

在这里需要进一步探讨异步 I/O 的实现。从前面过程分析可以知道，buffer I/O 不能实现异步 I/O，要避免进程阻塞，实现异步 I/O 必须使用 direct I/O。即使这样，异步 I/O 中仍然使用了自旋锁和信号量，进程仍然可能在某些地方被阻塞。

函数 mpage_bio_submit 提交 bio 的时候，要注册回调函数，以供中断处理函数中调用，如代码清单 10-13 所示。

代码清单 10-13　mpage_bio_submit（mpage.c）

```
static struct bio *mpage_bio_submit(int rw, struct bio *bio)
{
  bio->bi_end_io = mpage_end_io_read;
  if (rw == WRITE)
       bio->bi_end_io = mpage_end_io_write;
  submit_bio(rw, bio);
  return NULL;
}
```

mpage_bio_submit 函数提供了回调函数 mpage_end_io_read。这个函数是在 I/O 完成后的中断过程中调用的。I/O 中断在第 11 章分析，本章重点关注这个回调函数的实现，如代码清单 10-14 所示。

代码清单 10-14 mpage_end_io_read（mpage.c）

```
static int mpage_end_io_read(struct bio *bio, unsigned int bytes_done, int err)
{       /* 测试数据是否更新 */
  const int uptodate = test_bit(BIO_UPTODATE, &bio->bi_flags);
  struct bio_vec *bvec = bio->bi_io_vec + bio->bi_vcnt - 1;

  if (bio->bi_size)
       return 1;

  do {
       struct page *page = bvec->bv_page;

       if (--bvec >= bio->bi_io_vec)
           prefetchw(&bvec->bv_page->flags);

       if (uptodate) {
           /* 读成功，设置 page 为 update*/
           SetPageUptodate(page);
       } else {
           ClearPageUptodate(page);
           SetPageError(page);
       }
       /* 解锁 page ，唤醒被阻塞的读 page 进程 */
       unlock_page(page);
  } while (bvec >= bio->bi_io_vec);
  bio_put(bio);
  return 0;
}
```

函数 mpage_end_io_read 要遍历 bio 结构的每个向量，看相关页面是否获得最新的数据。如果成功，则设置页面状态为最新，同时解锁页面。这个解锁动作会唤醒等待该页面更新的进程，从之前阻塞点继续往下执行，如果数据未更新，则设置页面出错，然后解锁页面，唤醒等待页面更新的进程。回顾一下读页面的代码，此时进程被唤醒后，会检查页面状态是否为最新，如果不为最新，将设置 readpage_error 错误返回。

10.6 文件写过程代码分析

文件写的系统调用是 write。在内核中，文件的写过程是从 sys_write 函数开始，这个函数以及它调用的 vfs_write 和文件的读过程非常类似，就不再赘述。本节从文件系统提供的 write 函数开始分析。

1. generic_file_write 函数

ext2 文件系统提供的 write 函数是 generic_file_write，这个函数也是个通用函数，被很多文件系统所使用，如代码清单 10-15 所示。

代码清单 10-15 generic_file_write (filemap.c)

```
ssize_t generic_file_write(struct file *file, const char __user *buf,
              size_t count, loff_t *ppos)
{
  struct address_space *mapping = file->f_mapping;
  struct inode *inode = mapping->host;
  ssize_t  ret;
  struct iovec local_iov = { .iov_base = (void __user *)buf,
                             .iov_len = count };

  mutex_lock(&inode->i_mutex);
  ret = __generic_file_write_nolock(file, &local_iov, 1, ppos);
  mutex_unlock(&inode->i_mutex);
  /* 如果文件有 SYNC 标志，或者 inode 有 SYNC 标志 */
  if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
      ssize_t err;
      err = sync_page_range(inode, mapping, *ppos - ret, ret);
```

buffer I/O 的读写对象是 page cache，文件写只把数据写到 page cache 就返回。但是某些时候，用户希望真正写到硬盘，文件就需要设置 SYNC 标志。设置 SYNC 标志后，系统将调用 sync_page_range 把 page cache 的页面写入硬盘，这部分在第 12 章中分析，此处略过。

2. generic_file_buffered_write 函数

__generic_file_write_nolock 类似文件的读函数，它调用 __generic_file_aio_write_nolock，这个函数和读函数 __generic_file_aio_read 也很类似，所以此处略过。

__generic_file_aio_write_nolock 调用了 generic_file_buffered_write 函数执行写操作，我们从这里开始分析。generic_file_buffered_write 函数的代码如代码清单 10-16 所示。

代码清单 10-16 generic_file_buffered_write (filemap.c)

```
ssize_t
generic_file_buffered_write(struct kiocb *iocb, const struct iovec *iov,
      unsigned long nr_segs, loff_t pos, loff_t *ppos,
      size_t count, ssize_t written)
{
  struct file *file = iocb->ki_filp;
  struct address_space * mapping = file->f_mapping;
  /*mapping 的 a_ops 结构是初始化 inode 的时候由文件系统提供 */
  const struct address_space_operations *a_ops = mapping->a_ops;
  struct inode     *inode = mapping->host;
  long             status = 0;
  struct page      *page;
  struct page      *cached_page = NULL;
```

```
    size_t          bytes;
    struct pagevec  lru_pvec;
    const struct iovec *cur_iov = iov; /* current iovec */
    size_t          iov_base = 0;      /* offset in the current iovec */
    char __user     *buf;

    pagevec_init(&lru_pvec, 0);

    if (likely(nr_segs == 1))
        buf = iov->iov_base + written;
    else {
        filemap_set_next_iovec(&cur_iov, &iov_base, written);
        buf = cur_iov->iov_base + iov_base;
    }
```

generic_file_buffered_write 函数第一部分设置必要的参数。

文件的 f_mapping 成员是一个指向 address_space 结构的指针，而 address_space 结构包含了文件的读写操作函数。

输入参数 nr_segs 是段的数目。每个段代表独立的一段内存，文件读写时，可以指定多个段，由内核一次性处理。当前场景只有一个段，buf 参数指向保存写入数据的用户态内存的地址。

generic_file_buffered_write 函数随后部分检查需要写入的每个页面的状态，代码如下：

```
do {
    unsigned long index;
    unsigned long offset;
    size_t copied;
    offset = (pos & (PAGE_CACHE_SIZE -1)); /* Within page */
    index = pos >> PAGE_CACHE_SHIFT;
    bytes = PAGE_CACHE_SIZE - offset;

    /* Limit the size of the copy to the caller's write size */
    bytes = min(bytes, count);
    bytes = min(bytes, cur_iov->iov_len - iov_base);

    fault_in_pages_readable(buf, bytes);
    /* 从 page cache 申请一个页面 */
    page = __grab_cache_page(mapping,index,&cached_page,&lru_pvec);
    ……/* 省略部分代码 */
    /* 如果要写入的字节为 0，跳转 zero_length_segment*/
    if (unlikely(bytes == 0)) {
        status = 0;
        copied = 0;
        goto zero_length_segment;
    }

    status = a_ops->prepare_write(file, page, offset, offset+bytes);
    ……/* 省略部分代码 */
```

```
    /* 复制用户内存到 page cache 的页面 */
    if (likely(nr_segs == 1))
        copied = filemap_copy_from_user(page, offset, buf, bytes);
    else
        copied = filemap_copy_from_user_iovec(page, offset,
                        cur_iov, iov_base, bytes);
    flush_dcache_page(page);
    status = a_ops->commit_write(file, page, offset, offset+bytes);
    if (status == AOP_TRUNCATED_PAGE) {
    /* 如果这个页面无效了，需要重来一次 */
    page_cache_release(page);
    continue;
}
```

generic_file_buffered_write 函数第二部分循环遍历要写入数据的每一个页面。

变量 offset 是当前页面内的偏移值，bytes 是要写入的字节数，index 计算当前页面的索引值。以页面索引值为参数，向 page cache 申请页面，如果页面存在，则锁定页面，禁止其他任务再使用文件的这个页面。如果页面不存在，需要创建一个页面，加入 page cache，然后锁定页面。

锁定当前页面后，首先调用文件系统的 prepare_write 函数检查当前页内的每个文件块，然后将数据从用户状态缓存复制到 page cache，最后调用文件系统的 commit_write 再次检查当前页的每个文件块并修改块和页面的状态。

返回值 AOP_TRUNCATED_PAGE 代表所操作的 page cache 页面无效，这种情况文件位置和页面索引都不改变，重新尝试申请 page cache 页面，重复上述写入的过程。

generic_file_buffered_write 函数随后部分检查需要写入的字节数，代码如下：

```
zero_length_segment:
        if (likely(copied >= 0)) {
            /* 根据复制的字节数目，计算剩余的字节数和当前位置 */
            if (!status)
                status = copied;

            if (status >= 0) {
                written += status;
                count -= status;
                pos += status;
                buf += status;
                if (unlikely(nr_segs > 1)) {
                    filemap_set_next_iovec(&cur_iov, &iov_base, status);
                    if (count)
                        buf = cur_iov->iov_base + iov_base;
                } else {
                    iov_base += status;
                }
            }
        }
        if (unlikely(copied != bytes))
```

```
            if (status >= 0)
                status = -EFAULT;
        unlock_page(page);
        mark_page_accessed(page);
        page_cache_release(page);
        if (status < 0)
            break;
        /* 检查是否需要真正写硬盘 */
        balance_dirty_pages_ratelimited(mapping);
        cond_resched();
    } while (count);
```

generic_file_buffered_write 函数第三部分是 zero_length_segment 分支，也是当前页写入完成，进入下一个页面之前的参数调整部分。

如果当前页面已经成功写入 page cache，文件位置 pos，用户态内存都要加上已经完成的字节数，而需要写入的字节数 count 则减去已经完成的字节。

balance_dirty_pages_ratelimited 函数的作用是检查是否触发了内核的回写策略，是否需要将写入 page cache 的数据真正写入硬盘，这个函数将在第 12 章进行分析。

generic_file_buffered_write 函数随后检查文件的特殊标志，代码如下：

```
    *ppos = pos;
    if (cached_page)
        page_cache_release(cached_page);

    if (likely(status >= 0)) {
        if (unlikely((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
            if (!a_ops->writepage || !is_sync_kiocb(iocb))
                status = generic_osync_inode(inode, mapping,
                        OSYNC_METADATA|OSYNC_DATA);
        }
    }
   ....../* 省略 direct I/O 的代码 */
    pagevec_lru_add(&lru_pvec);
    return written ? written : status;
}
```

generic_file_buffered_write 函数第四部分检查文件是否具有 SYNC 标志。通常文件数据写到 page cache 就结束了，何时从 page cache 真正写入硬盘由内核的回写机制控制。但是如果文件具有 SYNC 标志或者文件系统 mount 时候设置了 MS_SYNCHRONOUS 标志，立即将修改的内容同步到硬盘并等待写入数据完成。

最后，generic_file_buffered_write 函数返回最终写入 page cache 的总字节数。

3. 获得文件块的物理块号

通过文件写数据到硬盘，必须获得文件块的物理块号，才能真正执行数据写入硬盘。这是文件系统提供的 prepare_write 函数和 commmit_write 函数的功能。对 ext2 文件系统而言，它提供的 prepare_write 函数是 ext2_prepare_write，commit_write 函数是 generic_commit_write。

ext2_prepare_write 是个封装函数，它真正调用的是 __block_prepare_write 函数。该函数要逐个获得页面内文件块的物理块号并检查它的状态是否最新，如代码清单 10-17 所示。

代码清单 10-17 __block_prepare_write(buffer.c)

```
static int __block_prepare_write(struct inode *inode, struct page *page,
     unsigned from, unsigned to, get_block_t *get_block)
{
  ……/* 省略部分代码 */
  blocksize = 1 << inode->i_blkbits;
  /* 为页面创建块缓存 */
  if (!page_has_buffers(page))
      create_empty_buffers(page, blocksize, 0);
  head = page_buffers(page);

  bbits = inode->i_blkbits;
 /* 计算页面内第一个块在文件内的块号 */
  block = (sector_t)page->index << (PAGE_CACHE_SHIFT - bbits);
```

__block_prepare_write 函数第一部分为页面创建 buffer head 管理结构，然后计算需要用到的参数。

变量 blocksize 保存文件块的大小，block 计算页面内第一个块在文件内的逻辑块号。

__block_prepare_write 函数随后检查所有需要写入的文件块，代码如下：

```
/* 循环遍历所有的块 */
for(bh = head, block_start = 0; bh != head || !block_start;
    block++, block_start=block_end, bh = bh->b_this_page) {
     block_end = block_start + blocksize;
     /* 如果块地址不在写的范围内，则进入下一个块 */
     if (block_end <= from || block_start >= to) {
         if (PageUptodate(page)) {
             if (!buffer_uptodate(bh))
                 set_buffer_uptodate(bh);
         }
         continue;
     }
     if (buffer_new(bh))
         clear_buffer_new(bh);
     /* 文件块未映射到硬盘 */
     if (!buffer_mapped(bh)) {
         WARN_ON(bh->b_size != blocksize);
         /* 获得文件块的物理块号，映射到具体的物理设备 */
         err = get_block(inode, block, bh, 1);
         if (err)
             break;
         if (buffer_new(bh)) {
             unmap_underlying_metadata(bh->b_bdev, bh->b_blocknr);
             /* 如果页面已经是最新内容，那么设置块缓存为最新 */
             if (PageUptodate(page)) {
```

```
                set_buffer_uptodate(bh);
                continue;
            }
            /* 将写入范围之外的数据填充为 0*/
            if (block_end > to || block_start < from) {
                void *kaddr;
                kaddr = kmap_atomic(page, KM_USER0);
                if (block_end > to)
                    memset(kaddr+to, 0, block_end-to);
                if (block_start < from)
                    memset(kaddr+block_start,
                        0, from-block_start);
                flush_dcache_page(page);
                kunmap_atomic(kaddr, KM_USER0);
            }
            continue;
        }
    }
    if (PageUptodate(page)) {
        if (!buffer_uptodate(bh))
            set_buffer_uptodate(bh);
        continue;
    }
    /* 写入的地址没按照文件块对齐，需要读出文件内容 */
    if (!buffer_uptodate(bh) && !buffer_delay(bh) &&
        (block_start < from || block_end > to)) {
        ll_rw_block(READ, 1, &bh);
        *wait_bh++=bh;
    }
}
```

__block_prepare_write 函数第二部分逐个遍历页面内所有的文件块。

如果当前块不在写入的范围内，只顺便检查一下页面状态。页面状态为 PG_uptodate，说明块缓存的状态也必然为 BH_uptodate，因此设置块缓存的状态为最新。如果页面状态不是 PG_uptodate，跳转下一个文件块，并不试图获得文件块的物理块号。

如果当前块在写入的地址范围内并且还没映射到物理设备，则调用文件系统 get_block 完成到物理块号的映射。映射完成后，同样检查页面状态是否为 PG_uptodate，是的话则设置块缓存状态为 BH_uptodate。

如果最终块缓存不是最新内容，且写的地址和块边界未对齐，则需要先把该块内容读进来。这是因为硬盘这样的块设备必须以扇区为最小访问单元。如果写入的内容在一个扇区的中间位置，必须把整个扇区内容读出来，把要写的部分更新，复合成最新的内容，才能写入硬盘。

__block_prepare_write 函数随后部分检查是否需要等待前面的读请求，代码如下：

```
while(wait_bh > wait) {
    wait_on_buffer(*--wait_bh);
    if (!buffer_uptodate(*wait_bh))
```

```
            err = -EIO;
    }
    if (!err) {
        bh = head;
        /* 遍历块缓存，清除 new 标志 */
        do {
            if (buffer_new(bh))
                clear_buffer_new(bh);
        } while ((bh = bh->b_this_page) != head);
        return 0;
```

__block_prepare_write 函数第三部分检查前面的处理过程是否产生了读请求，有读请求必须等读完成，否则块缓存状态不是最新的，将产生一个错误。

ext2 文件系统提供的 commit_write 就是 generic_commit_write。它的作用是逐个遍历所有的文件块，检查块缓存是否最新。如果块缓存是最新内容，标记文件块缓存为 uptodate，同时设置块缓存为 dirty，标记着块缓存的内容需要写入硬盘。

如果所有的文件块缓存都是最新的，标记整个页面为 uptodate。

文件写有可能导致文件长度变化。如果文件长度变化，则需要修改文件的长度。

10.7 本章小结

文件读写的过程比较复杂，涉及文件中一些复杂参数的计算和 page cache 中页面缓存的状态处理。读者可以将复杂问题形象化，自己设计一个文件，设置它的长度和需要读写的字节位置，对照代码进行推演，这样可以比较直观地理解文件的读写过程。

第 11 章

通用块层和 scsi 层

在内核中通用块层和 scsi 层的位置，上接文件系统的 VFS 层，下接硬盘驱动。通用块层的作用就是处理 I/O 的合并或者排序。而 scsi 层的作用主要是管理 scsi 设备、处理的设备的上线和离线、为设备加载合适的驱动等。scsi 层同时是通用块层的下一层，I/O 处理的时候也需要经过 scsi 层。通用块层和 scsi 层的一部分共同执行 I/O 的处理过程。

scsi 层在内核中担当特殊的角色。即使某些不是 scsi 设备的磁盘（比如 ATA 格式的硬盘），在内核中也是通过 scsi 层来管理的。本章关注重点是 I/O 的处理过程，主要涉及通用块层和 scsi 层的一部分，scsi 层对设备的管理功能不在本章讨论范围之内。

11.1 块设备队列

第 9 章分析了块设备的队列和队列处理函数。在 Linux 内核，读写操作是以一个个请求的方式出入块设备的队列。一个请求可以代表一个 I/O，也可以代表多个 I/O。如果一个 I/O 和其他 I/O 发生了合并，这两个 I/O 在一个请求里面。所有需要执行的请求，都要链接到块设备队列的链表。第 9 章还分析了块设备队列的电梯对象，电梯提供了块设备排序的算法和排序结构，本章将继续讨论。

11.1.1 scsi 块设备队列处理函数

scsi 作为一个广泛使用的框架，提供了一系列的块设备队列处理函数。这是在 scsi_alloc_queue 函数中实现的，如代码清单 11-1 所示。

代码清单 11-1 scsi_alloc_queue 函数

```
struct request_queue *scsi_alloc_queue(struct scsi_device *sdev)
{
  struct Scsi_Host *shost = sdev->host;
  struct request_queue *q;
```

```
    q = blk_init_queue(scsi_request_fn, NULL);
    if (!q)
        return NULL;

    blk_queue_prep_rq(q, scsi_prep_fn);

    blk_queue_max_hw_segments(q, shost->sg_tablesize);
    blk_queue_max_phys_segments(q, SCSI_MAX_PHYS_SEGMENTS);
    blk_queue_max_sectors(q, shost->max_sectors);
    blk_queue_bounce_limit(q, scsi_calculate_bounce_limit(shost));
    blk_queue_segment_boundary(q, shost->dma_boundary);
    blk_queue_issue_flush_fn(q, scsi_issue_flush_fn);
    blk_queue_softirq_done(q, scsi_softirq_done);

    if (!shost->use_clustering)
        clear_bit(QUEUE_FLAG_CLUSTER, &q->queue_flags);
    return q;
}
```

scsi_alloc_queue 函数为 scsi 块设备创建队列结构时调用。默认为 scsi 设备提供了出队列函数 scsi_request_fn 和软中断完成函数 scsi_softirq_done。

11.1.2 电梯算法和对象

电梯算法在内核中已经被抽象为一个对象，只要实现一些基本的电梯函数，就可以提供一个电梯算法。电梯结构的定义如代码清单 11-2 所示。

代码清单 11-2 elevator_noop 算法

```
static struct elevator_type elevator_noop = {
    .ops = {
        .elevator_merge_req_fn          = noop_merged_requests,
        .elevator_dispatch_fn           = noop_dispatch,
        .elevator_add_req_fn            = noop_add_request,
        .elevator_queue_empty_fn        = noop_queue_empty,
        .elevator_former_req_fn         = noop_former_request,
        .elevator_latter_req_fn         = noop_latter_request,
        .elevator_init_fn               = noop_init_queue,
        .elevator_exit_fn               = noop_exit_queue,
    },
    .elevator_name = "noop",
    .elevator_owner = THIS_MODULE,
};
```

电梯结构 elevator_type 最重要的执行函数是 elevator_add_req_fn 和 elevator_dispatch_fn，分别用来向电梯加入一个请求和从电梯获得一个请求。而 elevator_merge_req_fn 则实现 I/O 的合并。在下文将看到这些函数的使用方式。

电梯算法一般需要维护一个队列，这个队列是为了对请求排序或者执行 I/O 合并。

11.2 硬盘HBA抽象层

HBA（Host Bus Adapter，主机总线适配器）通常用来连接计算机内部总线和存储系统。用来接入硬盘的设备，如果是一个PCI设备，它既是一个PCI设备，同时支持SCSI硬盘或者ATA硬盘，它就是一个HBA设备。

HBA设备的驱动，既是PCI驱动，同时又要管理和控制硬盘，所以它也可算作是硬盘驱动。scsi层最终要把scsi命令发送到硬盘的驱动层。硬盘驱动层是内核软件的最后一层，当硬盘驱动把命令发送到硬件后，就脱离了软件控制的范围。

为便于管理硬盘驱动，内核抽象出一个scsi_host_template对象，所有的硬盘驱动都要以这种方式提供。将scsi_host_template结构的定义简化后，如代码清单11-3所示。

代码清单11-3 scsi_host_template函数

```
struct scsi_host_template {
  const char *name;

  int (* queuecommand)(struct scsi_cmnd *, void (*done)(struct scsi_cmnd *));
  int (* eh_abort_handler)(struct scsi_cmnd *);
  int (* eh_device_reset_handler)(struct scsi_cmnd *);
  ......
  int (* eh_host_reset_handler)(struct scsi_cmnd *);

};
```

对scsi_host_template结构的重要成员做如下解释。

- name：HBA卡的名字。
- queuecommand：I/O函数。通过这个函数将scsi命令发送到HBA卡，完成一次I/O。
- eh_abort_handler：撤销一个scsi命令。
- eh_device_reset_handler：reset某个硬盘设备。这个硬盘设备将离线，然后重新上线。
- eh_host_reset_handler：reset整个适配器芯片。执行这个调用，整个适配器芯片被重启，所有的硬盘离线，然后重新被扫描一次。

在scsi_host_template结构提供的调用函数中，异常处理占了很大一部分。对一个I/O来说，执行结果可以分为几种情况。

- I/O命令执行完成，而且I/O执行成功。
- I/O命令执行完成，但是I/O未成功，返回有错误。
- I/O超时未返回。

对于I/O返回有错误的情况，内核根据错误类型，选择再次执行该I/O命令或者交给scsi错误处理任务处理。对I/O超时未返回，则必须由scsi错误处理任务控制。

scsi错误处理任务通常首先abort（取消）出错的命令，如果不能奏效，尝试reset硬盘设备；如果仍然不能奏效，尝试reset总线；如果仍不能奏效，尝试reset整个芯片。由于各个厂家的HBA芯片实现各有不同，笔者发现很多情况下，这个错误处理流程会导致错误。最

常见的错误就是 CPU 占有率 100% 和 abort 处理时异常，导致硬盘离线。所以实现用户定制的错误处理逻辑可能是一个正确的选择。

11.3 I/O 的顺序控制

I/O 的顺序是通用块层中一个比较重要的概念。很多应用要求 I/O 必须按照指定的顺序执行，为此内核使用了 barrier I/O 的概念来实现 I/O 的顺序。指的是 barrier I/O 之前的 I/O 必须执行完毕，然后再执行 barrier I/O，而 barrier I/O 之后的 I/O 必须在 barrier I/O 执行完毕才能执行，就像 I/O 队列中插入了一个栅栏。

假设有 1，2，3，4，5 五个 I/O，其中 4 是 barrier I/O。这意味着，必须 1，2，3 执行完毕再执行 4，在 4 执行完毕之前不能执行任何 I/O。

需要指出的是，按照顺序下发 1，2，3，4，5 并不能保证 I/O 的完成顺序。这是因为硬盘本身有缓存（cache）和队列，并不是按照下发的顺序来执行 I/O。如何实现 I/O 顺序？

这就必须利用同步 cache 命令。即发现 4 是一个 barrier I/O 后，插一个同步 cache 命令，再下发 4，然后再插一个同步 cache 命令。在 4 执行完毕之前，不能再发送新的 I/O 命令。有人可能有疑问，1，2，3 的执行顺序能否保证？答案是不能，1，2，3 既然没设置 barrier 标志，意味可以乱序执行。

复杂一点的情况是，如果硬件支持 FUA 标志，也就是说这个 I/O 会跳过硬盘的 buffer，后面的一次同步 cache 命令就可以省略。更复杂的情况，如果硬件支持 TAG 队列功能，在执行保证顺序操作的过程中，仍然可以下发新的 I/O。在后面的代码中，可以见到对 I/O 的这种处理方法。

11.4 I/O 调度算法

一个 I/O 调度算法，关键就是实现 elevator 结构需要的几个函数，然后注册调度算法。用户可以通过 proc 文件系统选择 I/O 调度算法，内核将根据设定的 I/O 调度算法对 I/O 执行调度。

11.4.1 noop 调度算法

noop 调度算法是最简单的调度算法，如它的名字所言，基本什么都没做，等同于一个先进先出的调度算法，本身没对 I/O 进行真正的排序。I/O 调度算法的入队列函数决定了 I/O 是如何插入电梯队列的，因此首先分析入队列函数。noop 调度算法的入队列函数是 noop_add_request，如代码清单 11-4 所示。

代码清单 11-4 noop_add_request 函数

```
static void noop_add_request(request_queue_t *q, struct request *rq)
{
```

```
  struct noop_data *nd = q->elevator->elevator_data;
  list_add_tail(&rq->queuelist, &nd->queue);
}
```

noop 调度算法的入队列函数极其简单，就是把请求加入 noop 的链表，没有执行任何排序工作。

I/O 调度算法的出队列函数决定了 I/O 是如何被挑选，然后由硬盘驱动执行。noop 调度算法的出队列函数是 noop_dispatch，如代码清单 11-5 所示。

代码清单 11-5 noop_dispatch 函数

```
static int noop_dispatch(request_queue_t *q, int force)
{
  struct noop_data *nd = q->elevator->elevator_data;
 /* 判断队列非空 */
  if (!list_empty(&nd->queue)) {
      struct request *rq;
      /* 取出 I/O 请求 */
      rq = list_entry(nd->queue.next, struct request, queuelist);
      list_del_init(&rq->queuelist);
      /*I/O 请求送到块设备的队列 */
      elv_dispatch_sort(q, rq);
      return 1;
  }
  return 0;
}
```

noop 出队列函数从队列头取出最先插入队列的一个 I/O，然后将 I/O 送到块设备的队列。I/O 从块设备队列到驱动执行的过程，是由 scsi 层控制的。

11.4.2 deadline 调度算法

和 noop 调度算法相比，dealline 调度算法更复杂，它内部启用了一个红黑树结构来对 I/O 进行排序，保证 I/O 出队列时，已经按照扇区地址排好顺序了。deadline 调度算法代码在 block 目录的 dealline-iosched 文件中。在分析算法前，首先分析 deadline 调度算法定义的内部数据结构，如代码清单 11-6 所示。

代码清单 11-6 deadline_data 函数

```
struct deadline_data {
  struct rb_root sort_list[2];
  struct list_head fifo_list[2];

  struct deadline_rq *next_drq[2];
  struct hlist_head *hash;        /* request hash */
  unsigned int batching;          /* number of sequential requests made */
  sector_t last_sector;           /* head position */
  unsigned int starved;           /* times reads have starved writes */
```

```
  int fifo_expire[2];
  int fifo_batch;
  int writes_starved;
  int front_merges;

  mempool_t *drq_pool;
}
```

从 deadline 的数据结构可以发现，有两个队列，一个是红黑树 sort_list，另一个是链表 fifo_list。

一个 I/O 进入 dealline 队列的时候，要被插入红黑树队列和 fifo 两个队列，红黑树是按照扇区地址排序的队列，而 fifo 则按照 I/O 的先后顺序排序。每个队列都是两成员的数组，这是为了分别保存读请求和写请求，即读写请求分别保存在不同的队列中。

deadline 调度算法源文件只有几百行，重点分析插入队列的过程和出队列的过程，就可以基本明白 deadline 算法的设计思路。首先从入队列函数开始分析，deadline 算法的入队列函数是 deadline_add_request，如代码清单 11-7 所示。

代码清单 11-7 deadline_add_request 函数

```
static void
deadline_add_request(struct request_queue *q, struct request *rq)
{
  struct deadline_data *dd = q->elevator->elevator_data;
  struct deadline_rq *drq = RQ_DATA(rq);

  const int data_dir = rq_data_dir(drq->request);
/*deadline 请求加入红黑树队列 */
  deadline_add_drq_rb(dd, drq);
  /* 设置 deadline 请求的超时时间，然后加入到 fifo 队列 */
drq->expires = jiffies + dd->fifo_expire[data_dir];
  list_add_tail(&drq->fifo, &dd->fifo_list[data_dir]);
 /* 如果请求可以合并，则还要加入 hash 链表 */
  if (rq_mergeable(rq))
      deadline_add_drq_hash(dd, drq);
}
```

deadline_add_request 函数首先把请求加入红黑树队列。deadline 算法的红黑树根据扇区地址的顺序排序，因此插入过程是以 I/O 请求的扇区地址作为 key。其次要设置请求的超时时间，然后加入先进先出的 fifo 队列。设置超时时间的目的是防止 I/O 在队列中时间过长，影响业务的使用。

deadline 调度算法 I/O 出队列的函数是 deadline_dispatch_requests，如代码清单 11-8 所示。

代码清单 11-8 deadline_dispatch_requests 函数

```
static int deadline_dispatch_requests(request_queue_t *q, int force)
```

```
{
  struct deadline_data *dd = q->elevator->elevator_data;
  const int reads = !list_empty(&dd->fifo_list[READ]);
  const int writes = !list_empty(&dd->fifo_list[WRITE]);
  struct deadline_rq *drq;
  int data_dir;

  if (dd->next_drq[WRITE])
      drq = dd->next_drq[WRITE];
  else
      drq = dd->next_drq[READ];

  if (drq) {
      /* we have a "next request" */
      /*I/O 批处理 */
      if (dd->last_sector != drq->request->sector)
          /* end the batch on a non sequential request */
          dd->batching += dd->fifo_batch;

      if (dd->batching < dd->fifo_batch)
          /* we are still entitled to batch */
          goto dispatch_request;
  }
```

deadline 算法首先根据一系列设定的条件选择要处理的 I/O。

第一部分判断是否连续的批模式。如果当前 I/O 和前面 I/O 的扇区地址是连续的而非随机的，满足批模式的条件，则直接进入 dispatch_request 优先处理。deadline 还设置了一个 fifo_batch 数值，批模式优先处理的 I/O 个数不能超过这个数值。

```
if (reads) {
    BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[READ]));
    /* 存在读写请求，而写请求已经超过设定的饿死时间的情况下，优先处理写 */
    if (writes && (dd->starved++ >= dd->writes_starved))
        goto dispatch_writes;
    /* 如果不存在写超过饿死时间，则处理读请求 */
    data_dir = READ;
    goto dispatch_find_request;
}

if (writes) {
dispatch_writes:
    BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[WRITE]));
    dd->starved = 0;
    data_dir = WRITE;
    goto dispatch_find_request;
}
return 0;
```

deadline 算法第二部分检查读写请求的饥饿时间。deadline 算法设定读请求优先，如果

队列中存在读请求，则进入 dispatch_find_request 分支去挑选一个读请求进行处理，但是也不能无限制拖延写请求，每个写请求设定了一个饥饿时间，超过这个时间就必须优先处理写请求。

如果队列中没有读请求，而且不存在写请求超时，进入 dispatch_find_request 分支去挑选一个写请求进行处理。

```
dispatch_find_request:
  if (deadline_check_fifo(dd, data_dir)) {
      /* 如果有读请求超时，则优先处理 */
      /* An expired request exists - satisfy it */
      dd->batching = 0;
      drq = list_entry_fifo(dd->fifo_list[data_dir].next);

  } else if (dd->next_drq[data_dir]) {
      /* 从请求方向相同（都是读或者都是写）的队列挑选下一个请求 */
      drq = dd->next_drq[data_dir];
  } else {
      /* 从红黑树队列选一个请求 */
      dd->batching = 0;
      drq = deadline_find_first_drq(dd, data_dir);
  }

dispatch_request:
  dd->batching++;
  deadline_move_request(dd, drq);

  return 1;
```

deadline 算法第三部分要挑选一个出队列的 I/O。首先的条件是检查读请求是否超时。读请求入队列的时候设置了一个超时时间，超过这个时间必须马上处理。如果没超时的读请求，则从同方向的请求队列（都是读或者都是写）选择下一个请求。

最后，如果以上条件都不满足，说明同一个方向已经没有下一个请求，或者一个电梯已经完成（完成从低到高的一个循环），需要重新开始一轮循环，因此从红黑树队列中重新挑选一个新的 I/O。挑选 I/O 的函数是 deadline_find_first_drq，如代码清单 11-9 所示。

代码清单 11-9　deadline_find_first_drq 函数

```
static struct deadline_rq *
deadline_find_first_drq(struct deadline_data *dd, int data_dir)
{
  struct rb_node *n = dd->sort_list[data_dir].rb_node;

  for (;;) {
      if (n->rb_left == NULL)
          return rb_entry_drq(n);
      n = n->rb_left;
  }
}
```

deadline 算法重新挑选一个 I/O 请求的条件，就是根据 I/O 的方向（读或者写）从相应的红黑树队列中取出最左边的一个 I/O。因为硬盘顺序是按照从低到高的扇区地址排序的，选出最左边的 I/O，其实就是选择扇区最小的 I/O，后续 I/O 的扇区地址都大于这个 I/O，从而开启一轮新的循环。

11.5 I/O 的处理过程

11.5.1 I/O 插入队列的过程分析

根据前文第 10 章的分析，文件系统提交 I/O 都要通过 submit_bio 来提交一个 I/O。

1. submit_bio 函数

submit_bio 函数是文件系统 VFS 层和通用块层的衔接点，本节从这个函数开始分析，如代码清单 11-10 所示。

代码清单 11-10 submit_bio 函数

```
void submit_bio(int rw, struct bio *bio)
{
  /* 计算扇区数目 */
  int count = bio_sectors(bio);

  bio->bi_rw |= rw;
  if (rw & WRITE)
      count_vm_events(PGPGOUT, count);
  else
      count_vm_events(PGPGIN, count);
  /*dump io，用来调试 */
  if (unlikely(block_dump)) {
  ……/* 省略部分代码 */
  generic_make_request(bio);
}
```

2. generic_make_request 函数

submit_bio 函数调用 generic_make_request 向通用块层提交一个请求，输入的参数是一个 bio 结构，generic_make_request 函数要把 bio 转换为底层处理的请求结构，分两部分介绍。如代码清单 11-11 所示。

代码清单 11-11 generic_make_request (ll_rw_blk.c)

```
void generic_make_request(struct bio *bio)
{
  request_queue_t *q;
  sector_t maxsector;
  int ret, nr_sectors = bio_sectors(bio);
```

```
    dev_t old_dev;
    /* 在执行较长时间的任务之前，执行一次调度 */
    might_sleep();
    /* Test device or partition size, when known. */
    /* 检查是否超过了磁盘的扇区限制 */
    maxsector = bio->bi_bdev->bd_inode->i_size >> 9;
    if (maxsector) {
        sector_t sector = bio->bi_sector;
        if (maxsector < nr_sectors || maxsector - nr_sectors < sector) {
            handle_bad_sector(bio);
            goto end_io;
        }
    }
```

generic_make_request 函数第一部分检查最大扇区限制。执行 I/O 的起始扇区地址加 I/O 大小的结果不应该超过块设备的物理扇区地址，否则就要结束本次 I/O，返回错误。

generic_make_request 函数第二部分检查 I/O 的大小不应该超过块设备的最大可处理扇区。后者是块设备本身的特性，它限制了块设备一次 I/O 所能处理的最大扇区数目。

```
    maxsector = -1;
    old_dev = 0;
    do {
        char b[BDEVNAME_SIZE];
        /* 获得块设备的队列 */
        q = bdev_get_queue(bio->bi_bdev);
        if (!q) {
end_io:
            bio_endio(bio, bio->bi_size, -EIO);
            break;
        }

        /* I/O 超过了设备的物理最大允许扇区 */
        if (unlikely(bio_sectors(bio) > q->max_hw_sectors)) {
            goto end_io;
        }
        if (unlikely(test_bit(QUEUE_FLAG_DEAD, &q->queue_flags)))
            goto end_io;

        /* 如果磁盘有分区的处理 */
        blk_partition_remap(bio);
        ....../* 省略 I/O 追踪的代码 */
        maxsector = bio->bi_sector;
        old_dev = bio->bi_bdev->bd_dev;
        ret = q->make_request_fn(q, bio);
    } while (ret);
}
```

如果 I/O 所在的块设备是个分区块设备，必须找到分区的结构 hd_struct，I/O 的起始扇区地址要加上分区的起始地址，才是真正的物理地址。也要将分区块设备替换为块设备所在物理硬盘的主块设备。

3. __make_request 函数

最后，调用队列提供的 make_request_fn 函数。前文分析过，scsi 提供的队列 make_request_fn 就是 __make_request 函数，如代码清单 11-12 所示。

代码清单 11-12 __make_request (ll_rw_blk.c)

```
static int __make_request(request_queue_t *q, struct bio *bio)
{
  ....../* 省略部分代码 */
  sector = bio->bi_sector;
  nr_sectors = bio_sectors(bio);
  cur_nr_sectors = bio_cur_sectors(bio);
  prio = bio_prio(bio);

  rw = bio_data_dir(bio);
  sync = bio_sync(bio);
```

__make_request 函数是通用块层的主处理流程。它的作用就是判断 I/O 能否合并，如果可以合并，则不申请新请求，而是合入前面的请求；如果不能合并，就申请新请求结构，并且插入电梯的队列，分五部分介绍。

第一部分从 bio 结构获得 I/O 的起始扇区地址和以扇区度量的 I/O 大小。变量 sync 标志当前 I/O 是否需要同步处理。带有 sync 标志的 I/O 和普通 I/O 的处理方式有所不同。

注意

这里的 sync 标志和打开文件时候设置的 O_SYNC 标志不是一回事，两者的处理逻辑也不相同。

__make_request 函数第二部分首先检查是否需要 bounce。

```
blk_queue_bounce(q, &bio);
spin_lock_prefetch(q->queue_lock);
/* 检查是否 barrier I/O，后面要处理 */
barrier = bio_barrier(bio);
if (unlikely(barrier) && (q->next_ordered == QUEUE_ORDERED_NONE)) {
    err = -EOPNOTSUPP;
    goto end_io;
}
spin_lock_irq(q->queue_lock);
/* 如果是 barrier 请求，或者电梯空，不再判断是否需要合并 I/O*/
if (unlikely(barrier) || elv_queue_empty(q))
    goto get_rq;
```

为何需要 bounce？因为有些老设备支持的 DMA 区间不能覆盖内存空间，而 bio 结构成员 bio_vec 里面提供的内存可能不在设备能访问的内存范围之内，如果这种情况发生，就要另外申请低位内存，以低位内存作为 DMA 内存，等设备 DMA 操作完成后，再把数据复制到 bio_vec 提供的内存里。

其次检查 I/O 是否 barrier I/O。对于 barrier I/O，就直接进入 get_rq 分支，不再判断是否可以合并，因为 barrier I/O 的性质决定了它不能和之前的 I/O 进行合并。

__make_request 函数第三部分处理后向合并。

```
el_ret = elv_merge(q, &req, bio);
switch (el_ret) {
    /* 后向合并 */
    case ELEVATOR_BACK_MERGE:
        BUG_ON(!rq_mergeable(req));
        if (!q->back_merge_fn(q, req, bio))
            break;
        blk_add_trace_bio(q, bio, BLK_TA_BACKMERGE);

        /* 合并后，要改 bio 的尾部为新的这个 I/O，同时调整扇区数 */
        req->biotail->bi_next = bio;
        req->biotail = bio;
        req->nr_sectors = req->hard_nr_sectors += nr_sectors;
        req->ioprio = ioprio_best(req->ioprio, prio);
        drive_stat_acct(req, nr_sectors, 0);
        /* 调用电梯提供的合并函数 */
        if (!attempt_back_merge(q, req))
            elv_merged_request(q, req);
        goto out;
```

后向合并指的是当前 I/O 可以合并到某个请求的尾部，这种情况要把请求的尾部 bio 改为新的 bio，请求的扇区数要加上新 bio 的扇区数。然后调用电梯提供的合并函数执行 I/O 合并操作。

__make_request 函数第四部分处理前向合并。

```
/* 前向合并 */
case ELEVATOR_FRONT_MERGE:
  BUG_ON(!rq_mergeable(req));
  if (!q->front_merge_fn(q, req, bio))
      break;
  blk_add_trace_bio(q, bio, BLK_TA_FRONTMERGE);
  bio->bi_next = req->bio;

  req->bio = bio;
  req->buffer = bio_data(bio);
  req->current_nr_sectors = cur_nr_sectors;
  req->hard_cur_sectors = cur_nr_sectors;
  req->sector = req->hard_sector = sector;
  req->nr_sectors = req->hard_nr_sectors += nr_sectors;
  req->ioprio = ioprio_best(req->ioprio, prio);
  drive_stat_acct(req, nr_sectors, 0);
  if (!attempt_front_merge(q, req))
      elv_merged_request(q, req);
  goto out;
```

前向合并的处理方式和后向合并很相似，区别是前向合并要把新 bio 置于请求结构中 bio

链表的头部。然后调用电梯的合并函数执行 I/O 合并。

__make_request 函数第五部分申请请求。

```
get_rq:
  req = get_request_wait(q, rw, bio);
  init_request_from_bio(req, bio);
  spin_lock_irq(q->queue_lock);
 /* 如果块设备队列和电梯队列都空的，需要阻塞块设备队列 */
  if (elv_queue_empty(q))
      blk_plug_device(q);
      /* 把请求加入请求队列 */
      add_request(q, req);
      out:
  /* 如果是同步 I/O，立即解除队列阻塞，把 I/O 请求下发 */
  if (sync)
      __generic_unplug_device(q);
      spin_unlock_irq(q->queue_lock);
  return 0;
```

如果前面部分的前向合并和后向合并都不能执行，就必须申请一个新的请求。__make_request 函数调用 get_request_wait 申请一个新的请求，而 init_request_from_bio 函数则根据 bio 的类型初始化请求和设置请求标志。通常的请求标志如下列所示。

- REQ_FAILFAST：一般 I/O 失败后要重试几次。REQ_FAILFAST 标志不重试，立即返回。
- REQ_HARDBARRIER 和 REQ_SOFTBARRIER：标志一个 barrier 请求。
- REQ_RW_SYNC：同步标志。有同步标志，则立即对电梯队列执行 unplug 操作。

创建新请求后，要对队列执行 plug 和 unplug 操作。plug 就是阻塞，一个阻塞的队列是不能下发 I/O 的，要下发 I/O，必须执行 unplug。plug 队列的同时要启动一个定时器（默认 3 毫秒），在时间到达后 unplug 队列，开始下发 I/O。plug 队列和定时器的设置，说明 I/O 不是马上下发，需要等待后续的 I/O。对带有 sync 标志的 I/O 是个特例，要立即 unplug。

 **注意**

plug 和 unplug 针对的是块设备队列，操作对象并不一定是同一个 I/O。plug 时阻塞的 I/O，不一定在 unplug 时被下发，可能下发别的 I/O。

4. elv_merge 函数

在 __make_request 函数里面，判断两个 I/O 能否合并，使用了 elv_merge 函数，需要分析一下它的实现，函数代码如代码清单 11-13 所示。

代码清单 11-13 elv_merge 函数

```
int elv_merge(request_queue_t *q, struct request **req, struct bio *bio)
{
  elevator_t *e = q->elevator;
  int ret;
```

```
  /* 判断块设备队列能否合并? */
  if (q->last_merge) {
      ret = elv_try_merge(q->last_merge, bio);
      if (ret != ELEVATOR_NO_MERGE) {
          *req = q->last_merge;
          return ret;
      }
  }
  /* 否则在电梯队列里面查找，看能否合并 */
  if (e->ops->elevator_merge_fn)
      return e->ops->elevator_merge_fn(q, req, bio);
  return ELEVATOR_NO_MERGE;
}

static inline int elv_try_merge(struct request *__rq, struct bio *bio)
{
  int ret = ELEVATOR_NO_MERGE;
  if (elv_rq_merge_ok(__rq, bio)) {
      if (__rq->sector + __rq->nr_sectors == bio->bi_sector)
          ret = ELEVATOR_BACK_MERGE;
      else if (__rq->sector - bio_sectors(bio) == bio->bi_sector)
          ret = ELEVATOR_FRONT_MERGE;
  }
  return ret;
}
```

elv_merge 函数通过两步来判断 I/O 能否合并。

第一步判断能否和块设备队列的最后一个请求合并。判断是通过请求的扇区地址决定的，如果请求的最后扇区位置是 10000，而新的 I/O 从 10000 开始，那就是后向合并。如果新的 I/O 从 9995 开始，而 I/O 长度是五个扇区，那就是前向合并。

第二步调用电梯队列提供的函数判断能否合并。I/O 调度算法一般都要提供自己的合并函数。比如 deadline 调度算法提供的合并函数就要在电梯队列中寻找能和当前 I/O 合并的请求。

5. __elv_add_request 函数

对于不能合并的 I/O，需要把一个请求插入到队列。这通过函数 add_request 实现。它又是通过调用 __elv_add_request 函数实现插入的功能，因此直接分析 __elv_add_request 函数，它的代码如代码清单 11-14 所示。

代码清单 11-14 __elv_add_request 函数

```
void __elv_add_request(request_queue_t *q, struct request *rq,
                       int where, int plug)
{
  if (q->ordcolor)
      rq->flags |= REQ_ORDERED_COLOR;
```

```
    if (rq->flags & (REQ_SOFTBARRIER | REQ_HARDBARRIER)) {
        if (blk_barrier_rq(rq))
            q->ordcolor ^= 1;
        if (where == ELEVATOR_INSERT_SORT)
            where = ELEVATOR_INSERT_BACK;

        /* 如果是文件系统来的请求，则更新 end_sector 和边界请求 */
        if (blk_fs_request(rq)) {
            q->end_sector = rq_end_sector(rq);
            q->boundary_rq = rq;
        }
    } else if (!(rq->flags & REQ_ELVPRIV) && where == ELEVATOR_INSERT_SORT)
        where = ELEVATOR_INSERT_BACK;
    /* 阻塞块设备队列 */
    if (plug)
        blk_plug_device(q);

    elv_insert(q, rq, where);
}
```

I/O 插入电梯队列时，要判断 I/O 是否 barrier 请求。barrier I/O 要求保证顺序，因此 barrier I/O 之前的 I/O 请求必须完成，所以对于 barrier 请求，把默认插入方式改为从后方加入。

6. elv_insert 函数

函数最后调用了 elv_insert，这个函数要根据插入的位置执行，它的代码如代码清单 11-15 所示。

代码清单 11-15　elv_insert 函数

```
void elv_insert(request_queue_t *q, struct request *rq, int where)
{
  struct list_head *pos;
  unsigned ordseq;
  int unplug_it = 1;
  blk_add_trace_rq(q, rq, BLK_TA_INSERT);
  rq->q = q;

  switch (where) {
  case ELEVATOR_INSERT_FRONT:
  /* 对前向插入，标志加一个 SOFTBARRIER，请求直接加入块设备队列的头 */
      rq->flags |= REQ_SOFTBARRIER;
      list_add(&rq->queuelist, &q->queue_head);
      break;
```

elv_insert 函数第一部分是处理前向插入。前向插入意味着新请求位置在所有队列 I/O 的最前面。这种情况请求不需要加入电梯队列，因为它不需要在电梯中等待，而是直接插入到块设备队列的最前面。

elv_insert 函数第二部分是处理后向插入。后向插入意味着新请求位置在所有队列 I/O 的最后面。

```
case ELEVATOR_INSERT_BACK:
/* 后向插入，要排空队列中的 I/O。然后把请求加入块设备队列的链表尾 */
    rq->flags |= REQ_SOFTBARRIER;
    elv_drain_elevator(q);
    list_add_tail(&rq->queuelist, &q->queue_head);
    /* 移走阻塞标志，调用 request_fn 开始 I/O*/
    blk_remove_plug(q);
    q->request_fn(q);
    break;
```

现有电梯队列中的 I/O 怎么办？必须全部下发，所以首先要调用 elv_drain_elevator 函数排空电梯队列的所有 I/O，排空之后所有的 I/O 进入块设备队列，然后把请求加到块设备队列的尾部，实现后向插入的要求。最后 unplug 队列，调用队列的 request_fn 函数从块设备队列挑选 I/O 执行。

elv_insert 函数第三部分是处理按顺序插入和重插。

```
case ELEVATOR_INSERT_SORT:
    BUG_ON(!blk_fs_request(rq));
    rq->flags |= REQ_SORTED;
    q->nr_sorted++;
    /* 更新 last_merge，即最后一个可 merge 的请求 */
    if (q->last_merge == NULL && rq_mergeable(rq))
        q->last_merge = rq;
    q->elevator->ops->elevator_add_req_fn(q, rq);
    break;

case ELEVATOR_INSERT_REQUEUE:
    /* 重插队列 */
    rq->flags |= REQ_SOFTBARRIER;
    /* 如果队列无保证顺序操作，则插入队列头 */
    if (q->ordseq == 0) {
        list_add(&rq->queuelist, &q->queue_head);
        break;
    }
    /* 根据请求的性质，判断重插的位置 */
    ordseq = blk_ordered_req_seq(rq);

    list_for_each(pos, &q->queue_head) {
        struct request *pos_rq = list_entry_rq(pos);
        if (ordseq <= blk_ordered_req_seq(pos_rq))
            break;
    }

    list_add_tail(&rq->queuelist, pos);
    unplug_it = 0;
    break;
```

按顺序插入的情况比较简单，调用电梯队列提供的插入函数 elevator_add_req_fn 即可。

重插比较复杂，必须考虑 I/O 的顺序关系。重插入队列，是指 I/O 虽然返回，但是没有成功完成 I/O，需要重新插入电梯队列的情况。如果此时队列中不必保证顺序，把 I/O 简单地插入块设备队列头部即可。否则，就必须考虑 I/O 的位置。如果需要保证顺序，说明有 barrier I/O 需要处理，回顾前面分析的背景知识，barrier I/O 之前的 I/O 必须先完成，然后再完成 barrier I/O，barrier I/O 之后的 I/O 必须等 barrier I/O 完成之后再执行。ordseq 变量目的就是检查 I/O 的顺序，判断是 barrier I/O 之前的 I/O，还是之后的 I/O，或者是 barrier I/O 自身。根据这个顺序关系，逐个遍历整个队列，把 I/O 重新插入到正确的位置。

elv_insert 函数第四部分检查队列的限制。

```
/* 如果请求达到限制，则 unplug 队列 */
  if (unplug_it && blk_queue_plugged(q)) {
      int nrq = q->rq.count[READ] + q->rq.count[WRITE]
            - q->in_flight;

      if (nrq >= q->unplug_thresh)
            __generic_unplug_device(q);
  }
```

为了避免过多 I/O 请求在队列中堆积，队列设置了一个数值，当请求个数超过这个限制，就 unplug 队列，开始挑选 I/O 执行。

11.5.2 I/O 出队列的过程分析

什么条件下 I/O 从队列出来真正下发到硬盘？总结内核中的处理过程，列出以下几个条件。

- 第一个 I/O 启动了 3 毫秒的定时器，时间到了，会执行 unplug 函数，开始下发 I/O。
- 请求数目超过设定的限制（默认是 4），执行 unplug 函数，开始下发。
- 带有 sync 标志的 I/O，立即执行 unplug 函数，开始下发。
- barrier I/O，需要先清空电梯队列，然后执行 unplug 函数，开始下发。
- 当硬盘执行完毕一个 I/O，也要 unplug 队列，检查是否有 I/O 可以执行。

1. unplug 函数

I/O 的下发是通过 unplug 函数实现的，因此首先分析这个函数，它的代码如代码清单 11-16 所示。

代码清单 11-16 __generic_unplug_device

```
void __generic_unplug_device(request_queue_t *q)
{
  if (unlikely(blk_queue_stopped(q)))
      return;
  if (!blk_remove_plug(q))
      return;
  q->request_fn(q);
}
```

unplug 函数首先要检查队列是否设置了 stop 标志，设置这个标志将停止队列，不能进行 unplug 操作。其次清除队列的 plug 标志，然后调用队列的 request_fn 函数。

2. scsi_request_fn 函数

对于 scsi 设备而言，这个函数就是 scsi_request_fn，如代码清单 11-17 所示。

代码清单 11-17 scsi_request_fn 函数

```
static void scsi_request_fn(struct request_queue *q)
{
  ……/* 省略部分代码 */
  while (!blk_queue_plugged(q)) {
      int rtn;
      req = elv_next_request(q);
      /* 检查队列是否 ready，这个队列是块设备本身的设备队列，非上文的块设备队列 */
      if (!req || !scsi_dev_queue_ready(q, sdev))
          break;
      /* 设备状态是否 online*/
      if (unlikely(!scsi_device_online(sdev))) {
          scsi_kill_request(req, q);
          continue;
      }
```

scsi_request_fn 函数主体是个循环，它要一直执行直到队列为空或者 HBA 卡不能够再接收 I/O 为止。函数第一部分调用 elv_next_request 从块设备队列获得一个请求。然后检查块设备队列是否 ready 以及 scsi 设备状态是否 online。

scsi_request_fn 函数第二部分仍然检查各种异常情况，参考代码中添加的注释。如果无异常情况，把 I/O 请求从电梯队列中删除。

```
if (!(blk_queue_tagged(q) && !blk_queue_start_tag(q, req)))
    blkdev_dequeue_request(req);
sdev->device_busy++;
spin_unlock(q->queue_lock);
cmd = req->special;
/* 检查命令是否为空 */
if (unlikely(cmd == NULL)) {
    BUG();
}
spin_lock(shost->host_lock);
/*HBA 卡的 host 队列是否 ready*/
if (!scsi_host_queue_ready(q, shost, sdev))
    goto not_ready;
if (sdev->single_lun) {
    /* 检查 target 的用户是否就是 sdev 本身 */
    if (scsi_target(sdev)->starget_sdev_user &&
        scsi_target(sdev)->starget_sdev_user != sdev)
         goto not_ready;
    scsi_target(sdev)->starget_sdev_user = sdev;
}
```

```
shost->host_busy++;
spin_unlock_irq(shost->host_lock);
```

scsi_request_fn 函数第三部分，首先初始化保存 scsi 错误返回值的数据 buffer，然后调用 scsi_dispatch_cmd 将 I/O 下发到驱动。

```
        scsi_init_cmd_errh(cmd);
        rtn = scsi_dispatch_cmd(cmd);
        spin_lock_irq(q->queue_lock);
        if(rtn) {
            /* 出错，尝试重新 plug 队列 */
            if(sdev->device_busy == 0)
                blk_plug_device(q);
            break;
        }
}
goto out;
```

scsi_request_fn 函数第四部分是两个处理错误分支。

```
not_ready:
  spin_unlock_irq(shost->host_lock);
  spin_lock_irq(q->queue_lock);
  blk_requeue_request(q, req);
  sdev->device_busy--;
  if(sdev->device_busy == 0)
     blk_plug_device(q);
out:
  spin_unlock_irq(q->queue_lock);
  put_device(&sdev->sdev_gendev);
  spin_lock_irq(q->queue_lock);
}
```

not_ready 分支说明设备还没有准备好，因此把 I/O 重新插入电梯的队列。而 out 分支则只减少引用计数，不对 I/O 进行操作。这是因为进入 out 分支有两种情况，要么队列已经空，I/O 全部下放到驱动，要么 I/O 还没有真正从电梯队列解除，这两种情况都不需要将 I/O 重新插入电梯队列。

3. elv_next_request 函数

在 scsi_request_fn 函数中，获得一个 I/O 请求使用的是 elv_next_request 函数。这个函数从块设备队列获得一个 I/O，如代码清单 11-18 所示。

代码清单 11-18 elv_next_request 函数

```
struct request *elv_next_request(request_queue_t *q)
{
  struct request *rq;
  int ret;

  while ((rq = __elv_next_request(q)) != NULL) {
      if (!(rq->flags & REQ_STARTED)) {
```

```
            elevator_t *e = q->elevator;
            if (blk_sorted_rq(rq) && e->ops->elevator_activate_req_fn)
                e->ops->elevator_activate_req_fn(q, rq);
            rq->flags |= REQ_STARTED;
            blk_add_trace_rq(q, rq, BLK_TA_ISSUE);
        }
        /* 设置结束扇区 */
        if (!q->boundary_rq || q->boundary_rq == rq) {
            q->end_sector = rq_end_sector(rq);
            q->boundary_rq = NULL;
        }
```

elv_next_request 函数第一部分调用 __elv_next_request 从队列中获得一个 I/O 请求。如果该 I/O 未设置 REQ_STARTED 标志，说明设备驱动第一次见到这个请求，这种情况需要通知 I/O 调度算法。然后为该 I/O 设置 REQ_STARTED 标志。

如果队列的 boundary_rq 等于该 I/O，该 I/O 此时要离开队列，因此设置队列的 boundary_rq 为空并设置队列最终扇区地址为该 I/O 的最终扇区地址。如果队列的 boundary_rq 为空，执行同样的设置。

elv_next_request 函数第二部分调用队列的 prep_rq_fn 函数来预处理 I/O。根据预处理结果执行不同的处理。

```
if ((rq->flags & REQ_DONTPREP) || !q->prep_rq_fn)
  break;
 ret = q->prep_rq_fn(q, rq);
if (ret == BLKPREP_OK) {
  /*ok，返回请求，准备下发 */
  break;
} else if (ret == BLKPREP_DEFER) {
  /* 有错，保留请求在队列 */
  rq = NULL;
  break;
} else if (ret == BLKPREP_KILL) {
  /* 出错误了，结束这个请求 */
  int nr_bytes = rq->hard_nr_sectors << 9;
  if (!nr_bytes)
      nr_bytes = rq->data_len;

  blkdev_dequeue_request(rq);
  rq->flags |= REQ_QUIET;
  end_that_request_chunk(rq, 0, nr_bytes);
  end_that_request_last(rq, 0);
} else {
```

如果结果正确，则返回该 I/O，如果结果为 BLKPREP_DEFER，意味着部分正确，I/O 放在队列前面，但是不返回该 I/O，等待下次挑选 I/O 的动作。第一部分设置 REQ_STARTED 标志将阻止其他 I/O 越过该 I/O。如果结果为 BLKPREP_KILL，说明该 I/O 存在严重错误，需要从队列中删除并结束该 I/O。

真正从队列获取请求的函数是 __elv_next_request，它的代码如代码清单 11-19 所示。

代码清单 11-19 __elv_next_request

```
static inline struct request *__elv_next_request(request_queue_t *q)
{
  struct request *rq;
  while (1) {
      while (!list_empty(&q->queue_head)) {
          rq = list_entry_rq(q->queue_head.next);
          if (blk_do_ordered(q, &rq))
              return rq;
      }
      if (!q->elevator->ops->elevator_dispatch_fn(q, 0))
          return NULL;
  }
}
```

如果块设备队列不空，__elv_next_request 函数直接从块设备队列获得一个 I/O 请求，如果队列为空，则调用 elevator_dispatch_fn 函数从电梯队列获得一个请求，然后加入块设备队列的尾部。因为函数主体是一个循环，下一轮循环时，就可以从块设备队列获得请求了。

4. blk_do_ordered 函数

从队列获取一个请求后，要调用 blk_do_ordered 函数检查 I/O 的顺序，它的功能是为了处理 barrier I/O。如代码清单 11-20 所示。

代码清单 11-20 blk_do_ordered 函数

```
int blk_do_ordered(request_queue_t *q, struct request **rqp)
{
  struct request *rq = *rqp;
  /* 判断是否为 barrier I/O*/
  int is_barrier = blk_fs_request(rq) && blk_barrier_rq(rq);

  if (!q->ordseq) {
      /* 非 barrier 请求，返回 */
      if (!is_barrier)
          return 1;
  if (q->next_ordered != QUEUE_ORDERED_NONE) {
      *rqp = start_ordered(q, rq);
          return 1;
      } else {
          /* 如果队列切换到 ORDERED_NONE，清 I/O，错误标志设置为功能不支持 */
          blkdev_dequeue_request(rq);
          end_that_request_first(rq, -EOPNOTSUPP, rq->hard_nr_sectors);
          end_that_request_last(rq, -EOPNOTSUPP);
          *rqp = NULL;
          return 0;
      }
}
```

blk_do_ordered 函数第一部分检查之前在队列中是否设置了需要保证顺序的 I/O，如果没有且新挑选的 I/O 不是 barrier I/O，不需要保证顺序的处理，直接返回。如果该 I/O 是个 barrier I/O，调用 start_ordered 执行保证顺序的操作。如果队列切换为 ORDERED_NONE，说明队列不支持保证顺序的功能，直接清除 I/O 并返回。

blk_do_ordered 函数第二部分处理特殊情况，此时队列中已经存在需要保证顺序的 I/O。如果队列设置了 QUEUE_ORDERED_TAG 标志，说明硬件支持 TAG 队列，硬件可以对 I/O 执行顺序进行控制，只阻塞下一个 barrier I/O 就可以，不是 barrier I/O 的其他 I/O 可以放行，下发到驱动执行。

```
    if (!blk_fs_request(rq) && rq != &q->pre_flush_rq &&
                               rq != &q->post_flush_rq)
        return 1;

    if (q->ordered & QUEUE_ORDERED_TAG) {
        /* Ordered by tag.  Blocking the next barrier is enough. */
        if (is_barrier && rq != &q->bar_rq)
            *rqp = NULL;
    } else {
        /* Ordered by draining.  Wait for turn. */
        WARN_ON(blk_ordered_req_seq(rq) < blk_ordered_cur_seq(q));
        if (blk_ordered_req_seq(rq) > blk_ordered_cur_seq(q))
            *rqp = NULL;
    }
    return 1;
}
```

如果队列未设置 QUEUE_ORDERED_TAG 标志，需调用 blk_ordered_req_seq 检查 I/O 的顺序关系，判断 I/O 是否可以执行。如果 I/O 的顺序大于当前队列的处理顺序，说明 I/O 应该在后面处理，因此不应该被处理。I/O 顺序是 barrier I/O 必须要考虑的重要方面，当 I/O 没有成功完成，需要重新插入电梯队列的时候，也需要考虑 I/O 顺序，必须把 I/O 插入到正确的位置。

5. start_ordered 函数

start_ordered 函数根据要求设置同步 cache 命令，保证同步 cache 命令之前的 I/O 必须完成，它的代码如代码清单 11-21 所示。

代码清单 11-21　start_ordered 函数

```
static inline struct request *start_ordered(request_queue_t *q,
 struct request *rq)
{
  q->bi_size = 0;
  q->orderr = 0;
  q->ordered = q->next_ordered;
  /* 标记队列的保序操作开始 */
  q->ordseq |= QUEUE_ORDSEQ_STARTED;
```

```
    blkdev_dequeue_request(rq);
    q->orig_bar_rq = rq;
    rq = &q->bar_rq;
    rq_init(q, rq);
    rq->flags = bio_data_dir(q->orig_bar_rq->bio);
    /* 设备是否支持 FUA?*/
    rq->flags |= q->ordered & QUEUE_ORDERED_FUA ? REQ_FUA : 0;
    rq->elevator_private = NULL;
    rq->rl = NULL;
    init_request_from_bio(rq, q->orig_bar_rq->bio);
    rq->end_io = bar_end_io;
```

start_ordered 函数第一部分设置 I/O 请求的必要参数。由于这是一个 barrier I/O，所以将 I/O 保存在队列的 orig_bar_rq 成员后，使用了队列的 bar_rq 成员作为真正下发 I/O 的数据结构，并根据原来的 bio 结构初始化 bar_rq 成员的参数，对于 barrier I/O，它的 I/O 完成函数被设置为特殊提供的 bar_end_io 函数。

start_ordered 函数第二部分设置同步 cache 的命令。

```
    /* 是否需要后刷，如果需要，则增加一个同步 cache 的命令，插入块设备队列头 */
    if (q->ordered & QUEUE_ORDERED_POSTFLUSH)
        queue_flush(q, QUEUE_ORDERED_POSTFLUSH);
    else
        q->ordseq |= QUEUE_ORDSEQ_POSTFLUSH;

    /*barrier I/O 插入队列的头部 */
    elv_insert(q, rq, ELEVATOR_INSERT_FRONT);

    /* 是否需要前刷? 如果需要，增加一个同步 cache 命令，插入块设备队列头 */
    if (q->ordered & QUEUE_ORDERED_PREFLUSH) {
        queue_flush(q, QUEUE_ORDERED_PREFLUSH);
        rq = &q->pre_flush_rq;
    } else
        q->ordseq |= QUEUE_ORDSEQ_PREFLUSH;

    if ((q->ordered & QUEUE_ORDERED_TAG) || q->in_flight == 0)
        q->ordseq |= QUEUE_ORDSEQ_DRAIN;
    else
        rq = NULL;
    return rq;
}
```

如果队列带有 QUEUE_ORDERED_POSTFLUSH 标志，这种情况在队列头部插入一个同步 cache 命令。然后插入 barrier I/O 自身，最后如果队列带有 QUEUE_ORDERED_PREFLUSH 标志，说明要把之前下发的 I/O 执行完毕，还要插入一个同步 cache 命令，这个命令的目的是强制之前的 I/O 执行完毕。因为三次插入都是插入队列的头部，所以先插入 POSTFLUSH 同步 cache 命令，再插入 barrier I/O 自身，最后插入 PREFLUSH 同步 cache 命令。执行时，实际是 PREFLUSH 同步 cache 命令最先执行。

6. scsi_dispatch_cmd 函数

从电梯取得 I/O 请求后，要初始化请求的参数，然后检查队列和设备是否准备好。如果一切正常，调用 scsi_dispatch_cmd 发送 I/O 请求，如代码清单 11-22 所示。

代码清单 11-22 scsi_dispatch_cmd 函数

```
int scsi_dispatch_cmd(struct scsi_cmnd *cmd)
{
  struct Scsi_Host *host = cmd->device->host;
  unsigned long flags = 0;
  unsigned long timeout;
  int rtn = 0;
  /* 检查设备是否可用 */
  if (unlikely(cmd->device->sdev_state == SDEV_DEL)) {
      /* 如果设备不可用，则终结这个 I/O 命令 */
       cmd->result = DID_NO_CONNECT << 16;
       atomic_inc(&cmd->device->iorequest_cnt);
       __scsi_done(cmd);
       /* return 0 (because the command has been processed) */
       goto out;
  }

  /* 检查设备是否阻塞，如果阻塞，要重新把命令插入电梯队列 */
  if (unlikely(cmd->device->sdev_state == SDEV_BLOCK)) {
       scsi_queue_insert(cmd, SCSI_MLQUEUE_DEVICE_BUSY);
       SCSI_LOG_MLQUEUE(3, printk("queuecommand : device blocked \n"));
       goto out;
  }

  if (cmd->device->scsi_level <= SCSI_2 &&
      cmd->device->scsi_level != SCSI_UNKNOWN) {
      cmd->cmnd[1] = (cmd->cmnd[1] & 0x1f) | (cmd->device->lun << 5 & 0xe0);
  }

  /* 检查设备的 reset 时钟，避免设备未准备好的情况 */
  timeout = host->last_reset + MIN_RESET_DELAY;
  if (host->resetting && time_before(jiffies, timeout)) {
      int ticks_remaining = timeout - jiffies;
      while (--ticks_remaining >= 0)
          mdelay(1 + 999 / HZ);
      host->resetting = 0;
  }

  /* 设置该 scsi 命令的超时时间 */
  scsi_add_timer(cmd, cmd->timeout_per_command, scsi_times_out);
  scsi_log_send(cmd);
  atomic_inc(&cmd->device->iorequest_cnt);

  /* 检查命令是否超过 hba 支持的最大命令长度，超过结束命令 */
  if (CDB_SIZE(cmd) > cmd->device->host->max_cmd_len) {
```

```
        cmd->result = (DID_ABORT << 16);
        scsi_done(cmd);
        goto out;
    }

    spin_lock_irqsave(host->host_lock, flags);
    scsi_cmd_get_serial(host, cmd);

    if (unlikely(host->shost_state == SHOST_DEL)) {
        cmd->result = (DID_NO_CONNECT << 16);
        scsi_done(cmd);
    } else {
     /* 调用 HBA 提供的命令函数 */
        rtn = host->hostt->queuecommand(cmd, scsi_done);
    }
    spin_unlock_irqrestore(host->host_lock, flags);
    if (rtn) {/* 命令执行中出错，则删掉命令的计时器，然后把命令再次插入队列 */
        if (scsi_delete_timer(cmd)) {
            atomic_inc(&cmd->device->iodone_cnt);
            scsi_queue_insert(cmd,
                        (rtn == SCSI_MLQUEUE_DEVICE_BUSY) ?
                         rtn : SCSI_MLQUEUE_HOST_BUSY);
        }
```

scsi_dispatch_cmd 函数提供了很多注释，主要是处理很多设备不可用的情况。如果设备可用，则调用 hba 卡提供的 queuecommand 把 scsi 命令提交给硬盘驱动，同时提供了一个回调函数 scsi_done。

异常处理有多种情况，如果设备不可用，要停止 I/O，返回错误。如果是执行 I/O 中错误，而且符合重试条件，要把 I/O 重新插入电梯队列，再次执行 I/O。

11.5.3 I/O 返回路径

scsi_dispatch_cmd 提交 scsi 命令的时候，要提供一个回调函数 scsi_done。这个回调是在硬盘驱动的中断中调用的。scsi 命令从中断中返回，意味着这个命令已经执行完毕，有可能是成功执行了 scsi 命令，也有可能出现错误。不管成功还是错误，内核都需要处理这两种情况。

scsi_done 是 I/O 返回路径上的第一个函数，因此就从这个函数开始分析过程，它的代码如代码清单 11-23 所示。

代码清单 11-23 scsi_done 函数

```
static void scsi_done(struct scsi_cmnd *cmd)
{
  if (!scsi_delete_timer(cmd))
      return;
  __scsi_done(cmd);
}
```

scsi_done 函数首先要停止 scsi 命令的计时器，然后调用 __scsi_done。__scsi_done 函数也比较简单，如代码清单 11-24 所示。

代码清单 11-24 __scsi_done 函数

```
void __scsi_done(struct scsi_cmnd *cmd)
{
  struct request *rq = cmd->request;
  cmd->serial_number = 0;
  /* 将命令的完成次数加 1*/
  atomic_inc(&cmd->device->iodone_cnt);
  /* 如果 result 不为 0，错误计数加 1*/
  if (cmd->result)
      atomic_inc(&cmd->device->ioerr_cnt);
  rq->completion_data = cmd;
  blk_complete_request(rq);
}
```

中断返回函数的参数是一个 scsi 命令，__scsi_done 函数首先要从 scsi 命令结构中获得当初下发 I/O 时使用的请求对象，然后调用 blk_complete_request 函数来处理一个 I/O 的返回过程。

blk_complete_request 是中断上下文的最后一个函数，它要启动一个软中断继续处理 I/O，如代码清单 11-25 所示。

代码清单 11-25 blk_complete_request 函数

```
void blk_complete_request(struct request *req)
{
  struct list_head *cpu_list;
  unsigned long flags;

  BUG_ON(!req->q->softirq_done_fn);
  local_irq_save(flags);
  cpu_list = &__get_cpu_var(blk_cpu_done);
  /* 将请求加入每 cpu 变量链表 blk_cpu_done*/
  list_add_tail(&req->donelist, cpu_list);
  /* 启动一个软中断 */
  raise_softirq_irqoff(BLOCK_SOFTIRQ);
  local_irq_restore(flags);
}
```

截至目前，代码一直在硬盘驱动的中断上下文中执行，而中断上下文具有高优先级，会阻塞硬盘自身的中断，因此 blk_complete_request 函数启动块设备的软中断来处理中断的下半部。将中断处理代码划分为两部分，是内核的常见方式。网络设备的中断处理代码中，同样启动软中断来处理中断的下半部。

为了传递参数到软中断上下文，内核定义了一个 CPU 变量 blk_cpu_done，这是一个链表，返回 I/O 的请求结构要加入该链表。块设备软中断的处理函数是 blk_done_softirq。该函

数要从 CPU 变量 blk_cpu_done 获得已经完成的 I/O 请求，然后调用队列的软中断处理函数 softirq_done_fn 处理。前文已经分析过，这个软中断处理函数是 scsi_softirq_done。它的代码如代码清单 11-26 所示。

代码清单 11-26 scsi_softirq_done 函数

```
static void scsi_softirq_done(struct request *rq)
{
  struct scsi_cmnd *cmd = rq->completion_data;
  unsigned long wait_for = (cmd->allowed + 1) * cmd->timeout_per_command;
  int disposition;
 /* 初始化错误 I/O 链表 */
  INIT_LIST_HEAD(&cmd->eh_entry);
 /* 判断 I/O 是否成功完成 */
  disposition = scsi_decide_disposition(cmd);
  if (disposition != SUCCESS &&
     time_before(cmd->jiffies_at_alloc + wait_for, jiffies)) {
     /* 已经超时了，则设置命令完成，不再重复执行命令 */
     disposition = SUCCESS;
  }

  scsi_log_completion(cmd, disposition);
  switch (disposition) {
     case SUCCESS:
          scsi_finish_command(cmd);
          break;
     case NEEDS_RETRY:
          scsi_retry_command(cmd);
          break;
     case ADD_TO_MLQUEUE:
          scsi_queue_insert(cmd, SCSI_MLQUEUE_DEVICE_BUSY);
          break;
     default:
          if (!scsi_eh_scmd_add(cmd, 0))
               scsi_finish_command(cmd);
  }
}
```

软中断处理函数 scsi_softirq_done 根据命令执行的结果，分为四种情况来处理。1）结束命令，交给上层处理，这种情况并不一定命令成功完成了，如果命令超时了，也要结束命令，交给上层处理；2）重试命令；3）命令重新插入电梯队列，重新排队；4）由 scsi 错误处理线程来处理。

scsi_decide_disposition 函数值得仔细研究，这个函数总结了所有的 scsi 错误类型，对每种错误类型给出了处理措施。

如果 I/O 正常完成，调用 scsi_finish_command 函数返回上层，这个函数比较简单，它最终又调用了命令本身的 done 函数。done 函数是在初始化 scsi 命令时候调用 sd_init_command 函数设置为 sd_rw_intr，同时还设置了 scsi 命令的最大重试次数和超时时间值。

1. sd_rw_intr 函数

sd_rw_intr 函数要对 scsi 命令的各种错误进行处理，如代码清单 11-27 所示。

代码清单 11-27 sd_rw_intr 函数

```
static void sd_rw_intr(struct scsi_cmnd * SCpnt)
{
  int result = SCpnt->result;
  unsigned int xfer_size = SCpnt->request_bufflen;
  unsigned int good_bytes = result ? 0 : xfer_size;
  u64 start_lba = SCpnt->request->sector;
  u64 bad_lba;
  struct scsi_sense_hdr sshdr;
  int sense_valid = 0;
  int sense_deferred = 0;
  int info_valid;

  if (result) {
      /* 设置 scsi 命令的 sense_key 和 asc 值, ascq 值 */
      sense_valid = scsi_command_normalize_sense(SCpnt, &sshdr);
      if (sense_valid)
          sense_deferred = scsi_sense_is_deferred(&sshdr);
  }
  if (driver_byte(result) != DRIVER_SENSE &&
     (!sense_valid || sense_deferred))
      goto out;
```

第一部分，如果 scsi 命令的返回结果 result 不为 0，说明命令执行中产生了错误。scsi 协议定义了可能的错误类型，通过三个参数共同说明错误类型，这三个参数是 sense key、asc 和 acsq。参数的详细定义读者需要参考 scsi 协议文档，本书只对代码中处理的部分错误进行解释。

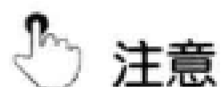
sd_rw_intr 函数第二部分主要是检查 scsi 命令的 sense key。

```
  switch (sshdr.sense_key) {
  case HARDWARE_ERROR:      /* 硬件错误 */
  case MEDIUM_ERROR:        /* 坏道 */
      if (!blk_fs_request(SCpnt->request))
          goto out;
      info_valid = scsi_get_sense_info_fld(SCpnt->sense_buffer,
                              SCSI_SENSE_BUFFERSIZE,
                              &bad_lba);
      if (!info_valid)
          goto out;
      if (xfer_size <= SCpnt->device->sector_size)
          goto out;
      switch (SCpnt->device->sector_size) {
      case 256:
          start_lba <<= 1;
          break;
```

```
        case 512:
            break;
        case 1024:
            start_lba >>= 1;
            break;
        case 2048:
            start_lba >>= 2;
            break;
        case 4096:
            start_lba >>= 3;
            break;
        default:
            /* Print something here with limiting frequency. */
            goto out;
            break;
        }
        good_bytes = (bad_lba - start_lba)*SCpnt->device->sector_size;
        break;
    case RECOVERED_ERROR:                /* 可修复的错误 */
    case NO_SENSE:                       /*no sense, 成功执行 */
        SCpnt->result = 0;
        memset(SCpnt->sense_buffer, 0, SCSI_SENSE_BUFFERSIZE);
        good_bytes = xfer_size;
        break;
    case ILLEGAL_REQUEST:                /* 不合法的请求 */
        if (SCpnt->device->use_10_for_rw &&
           (SCpnt->cmnd[0] == READ_10 ||
            SCpnt->cmnd[0] == WRITE_10))
            SCpnt->device->use_10_for_rw = 0;
        if (SCpnt->device->use_10_for_ms &&
           (SCpnt->cmnd[0] == MODE_SENSE_10 ||
            SCpnt->cmnd[0] == MODE_SELECT_10))
            SCpnt->device->use_10_for_ms = 0;
        break;
    default:
        break;
    }
  out:
    scsi_io_completion(SCpnt, good_bytes);
```

如果 sense key 不够，还需要一些附加的信息，通过 sense_buffer 里面的数据 asc 和 ascq 表示。sense key 主要包括以下几个。

- HARDWARE_ERROR：硬件错误。
- MEDIUM_ERROR：坏道错误。
- RECOVERED_ERROR：可修复的错误。
- NO_SENSE：无 sense 数据。
- ILLEGAL_REQUEST：不合法的请求。

 **注意**

RECOVERED_ERROR 和 NO_SENSE 代表命令已经成功执行了，这两个 sense key 作用只是提示用户。

变量 good_bytes 的含义有两种：1）如果命令成功完成，good_bytes 就是 scsi 命令设定的读写数值；2）如果命令部分完成（比如由于坏道错误只读了一部分数据），good_bytes 代表完成的字节数。

2. scsi_io_completion 函数

good_bytes 要作为输入参数调用 scsi_io_completion，由这个函数继续 I/O 的返回过程，如代码清单 11-28 所示。

代码清单 11-28 scsi_io_completion 函数

```
void scsi_io_completion(struct scsi_cmnd *cmd, unsigned int good_bytes)
{
  int result = cmd->result;
  int this_count = cmd->request_bufflen;
  request_queue_t *q = cmd->device->request_queue;
  struct request *req = cmd->request;
  int clear_errors = 1;
  struct scsi_sense_hdr sshdr;
  int sense_valid = 0;
  int sense_deferred = 0;

  scsi_release_buffers(cmd);
  if (result) {
      sense_valid = scsi_command_normalize_sense(cmd, &sshdr);
      if (sense_valid)
          sense_deferred = scsi_sense_is_deferred(&sshdr);
  }

 /*scsi 命令来自 SG_IO，也就是说不是来自于文件系统 */
  if (blk_pc_request(req)) { /* SG_IO ioctl from block level */
      req->errors = result;
      if (result) {
          clear_errors = 0;
          if (sense_valid && req->sense) {
              /*
               * SG_IO wants current and deferred errors
               */
              int len = 8 + cmd->sense_buffer[7];
              /* 复制返回的 sense buffer 内容 */
              if (len > SCSI_SENSE_BUFFERSIZE)
                  len = SCSI_SENSE_BUFFERSIZE;
              memcpy(req->sense, cmd->sense_buffer,  len);
              req->sense_len = len;
          }
```

```
        } else
            req->data_len = cmd->resid;
    }
```

scsi_io_completion 第一部分和 sd_rw_intr 函数一样，都根据 scsi 命令是否成功完成，将 sense key 和 asc、ascq 值复制出来。然后要检查 I/O 命令是否来自 SG_IO（SG_IO 指 scsi 命令来自于文件系统的 I/O control 调用，而不是来自文件系统本身）。这种情况要把返回命令的 sense buffer 整个复制出来。

scsi_io_completion 第二部分调用 scsi_end_request 函数返回上层处理。

```
if (clear_errors)
    req->errors = 0;
/* 命令成功完成返回，否则需要进一步处理 */
if (scsi_end_request(cmd, 1, good_bytes, result == 0) == NULL)
    return;
```

如果命令成功完成，整个 I/O 返回过程就执行完毕，如果命令没有成功完成，还需要第三部分处理错误。

命令是否成功完成，是由输入参数 good_bytes 决定的。

```
/* good_bytes = 0, or (inclusive) there were leftovers and
 * result = 0, so scsi_end_request couldn't retry.
 */
if (sense_valid && !sense_deferred) {
    switch (sshdr.sense_key) {
    case UNIT_ATTENTION:
         if (cmd->device->removable) {
             /* 如果设备被移走，则结束请求 */
             cmd->device->changed = 1;
             scsi_end_request(cmd, 0, this_count, 1);
             return;
        } else {
             /* 重新插入命令 */
             scsi_requeue_command(q, cmd);
             return;
        }
        break;
    case ILLEGAL_REQUEST:
        if ((cmd->device->use_10_for_rw &&
           sshdr.asc == 0x20 && sshdr.ascq == 0x00) &&
           (cmd->cmnd[0] == READ_10 ||
            cmd->cmnd[0] == WRITE_10)) {
            cmd->device->use_10_for_rw = 0;
            /* This will cause a retry with a
             * 6-byte command.
             */
            scsi_requeue_command(q, cmd);
            return;
        } else {
```

```
                scsi_end_request(cmd, 0, this_count, 1);
                return;
            }
            break;
        case NOT_READY:
            if (sshdr.asc == 0x04) {
                switch (sshdr.ascq) {
                case 0x01: /* becoming ready */
                case 0x04: /* format in progress */
                case 0x05: /* rebuild in progress */
                case 0x06: /* recalculation in progress */
                case 0x07: /* operation in progress */
                case 0x08: /* Long write in progress */
                case 0x09: /* self test in progress */
                    scsi_requeue_command(q, cmd);
                    return;
                default:
                    break;
                }
            }
            if (!(req->flags & REQ_QUIET)) {
                scmd_printk(KERN_INFO, cmd, "Device not ready: ");
                scsi_print_sense_hdr("", &sshdr);
            }
            scsi_end_request(cmd, 0, this_count, 1);
            return;
        case VOLUME_OVERFLOW:
            if (!(req->flags & REQ_QUIET)) {
                scmd_printk(KERN_INFO, cmd, "Volume overflow, CDB: ");
                __scsi_print_command(cmd->cmnd);
                scsi_print_sense("", cmd);
            }
            /* See SSC3rXX or current. */
            scsi_end_request(cmd, 0, this_count, 1);
            return;
        default:
            break;
        }
    }
    /* 如果返回 DID_RESET，则重新插入命令 */
    if (host_byte(result) == DID_RESET) {
        scsi_requeue_command(q, cmd);
        return;
    }
    if (result) {
        if (!(req->flags & REQ_QUIET)) {
        }
    }
    scsi_end_request(cmd, 0, this_count, !result);
}
```

scsi_io_completion 第三部分根据 sense key 和 asc、ascq 的值决定处理的方式。处理方式其实比较简单，一种是结束请求，另一种是把请求重新插入电梯队列。

值得注意的是，由于 scsi 硬件种类繁杂，错误定义混乱，scsi 返回信息很多时候不能准确反映设备的情况，这造成内核 I/O 处理的异常。比如有的 scsi 硬件总是返回 DID_RESET，结果 I/O 请求无限次重插，CPU 占有率达到 100%，阻塞了操作系统的运行。

3. scsi_end_request 函数

scsi_io_completion 函数分析完成后，还需要重点分析 scsi_end_request 函数，观察 I/O 的返回过程。它的代码如代码清单 11-29 所示。

代码清单 11-29 scsi_end_request 函数

```
static struct scsi_cmnd *scsi_end_request(struct scsi_cmnd *cmd, int uptodate,
                      int bytes, int requeue)
{
  request_queue_t *q = cmd->device->request_queue;
  struct request *req = cmd->request;
  unsigned long flags;

  /* 根据 uptodate 和 goodbytes 结束请求 */
  if (end_that_request_chunk(req, uptodate, bytes)) {
       /* 如果返回非 0，说明还有块未完成 */
       int leftover = (req->hard_nr_sectors << 9);

       if (blk_pc_request(req))
            leftover = req->data_len;

       /* kill remainder if no retrys */
       if (!uptodate && blk_noretry_request(req))
           /* 如果返回了 fastfail，说明不需要重试，则直接结束 */
           end_that_request_chunk(req, 0, leftover);
       else {
       /* 重插入队列 */
       if (requeue) {
           scsi_requeue_command(q, cmd);
           cmd = NULL;
       }
       return cmd;
    }
  }
  ……/* 省略部分代码 */
  scsi_next_command(cmd);
  return NULL;
}
```

scsi_end_request 函数调用 end_that_request_chunk 函数结束 I/O 的返回过程。如果返回结果不为 0，说明 I/O 没有成功完成，根据返回结果有两种处理方式。如果返回结果带有 REQ_FAILFAST 标志，指示快速返回，不需要重试，直接结束 I/O，否则就将 I/O 重新插入电梯队

列，再次执行。

最后调用 scsi_next_command 从队列中挑选下一个 I/O 执行。

参数 good_bytes 说明了 I/O 的完成情况，是部分完成还是全部完成。scsi_end_request 要根据 good_bytes 确定如何结束 I/O 请求。

end_that_request_chunk 调用了 __end_that_request_first 来完成 bio 的返回。回顾第 10 章的内容，bio 设置的回调函数是 mpage_end_io_read。__end_that_request_first 函数调用这个回调函数，完成唤醒阻塞进程的任务，通知 I/O 完成返回。

11.6 本章小结

通用块层、I/O 调度算法和 scsi 层，共同构建了文件系统之下的 I/O 处理流程。Linux 内核采用了很多对象的思想来设计程序，这样块设备的执行流程和电梯算法都对象化了，用户可以自行设计自己的模块来替换内核提供的软件模块。

第 12 章

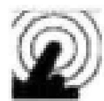
内核回写机制

回顾第 10 章关于文件写的部分，Linux 的写操作只是写数据到 page cache 中，真正的写磁盘要由内核的 pdflush 线程完成的。

12.1 内核的触发条件

何时启动写磁盘操作由内核根据一些条件触发 pdflush 线程完成。内核的触发条件如下。

- 文件系统的写函数 generic_file_buffered_write 要调用 balance_dirty_pages_ratelimited。后者首先检查累加的 dirty 页（这是一个 CPU 变量）是否达到一个限制，超过限制则执行写操作。
- 内核定时器触发。当内核启动的时候，要启动一个回写定时器（默认是 5 秒）。当定时时间到达，触发 pdflush 线程执行写操作。
- 当系统申请内存失败，或者文件系统执行同步（sync）操作，或者内存管理模块试图释放更多内存的时候，都可能触发 pdflush 线程回写页面

 提示

CPU 变量是内核的一种数据结构，每个 CPU 核都有变量的一个副本。

12.2 内核回写控制参数

内核通过 4 个参数控制数据回写的算法。这四个参数由 proc 文件系统提供，用户可以直接修改这些参数从而调整内核回写的行为。具体如表 12-1 所示。

表 12-1 内核回写控制参数

参数名称	参数位置	内核位置	功能描述
脏页比例	/proc/sys/vm/ dirty_ratio	page-writeback.c int vm_dirty_ratio = 40	百分比系数。如果脏页占比超过这个系数，当前进程开始回写脏数据。这种回写称为直接写
背景脏页比例	/proc/sys/vm/ dirty_background_ratio	page-writeback.c int dirty_background_ratio = 10	百分比系数。如果脏页占系统内存可分配内存超过这个系数，触发 pdflush 开始回写脏数据。这种回写称为背景写
写延时	/proc/sys/vm/ dirty_writeback_centisecs	page-writeback.c int dirty_writeback_centisecs = 5 × 100	控制 pdflush 的运行时间间隔。单位是 1/100 秒。默认数值是 500，也就是 5 秒。每 5 秒控制 pdflush 线程回写 page cache 的脏数据
最大 I/O 延时	/proc/sys/vm/ dirty_expire_centisecs	page-writeback.c int dirty_expire_centisecs = 30 × 100	脏页数据的最大延时时间。超过这个时间的脏数据，pdflush 线程要回写到磁盘。默认是 30 秒

12.3 定时器触发回写

系统初始化的 start_kernel 函数启动了一个定时器，这个定时器的作用就是推动内核回写数据。因此从 start_kernel 函数开始分析内核的回写机制。如代码清单 12-1 所示。

代码清单 12-1 start_kernel 函数

```
asmlinkage void __init start_kernel(void)
{
  ....../* 省略无关代码 */
  /* rootfs populating might need page-writeback */
  page_writeback_init();
  ....../* 省略无关代码 */
}
```

12.3.1 启动定时器

start_kennel 函数调用 page_writeback_init 函数启动定时器，如代码清单 12-2 所示。

代码清单 12-2 page_writeback_init 函数

```
void __init page_writeback_init(void)
{
  long buffer_pages = nr_free_buffer_pages();
  long correction;

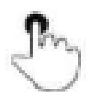
  total_pages = nr_free_pagecache_pages();
  correction = (100 * 4 * buffer_pages) / total_pages;
```

```
    if (correction < 100) {
        /* 脏页比例和背景脏页比例两个参数乘以内存比例 */
        dirty_background_ratio *= correction;
        dirty_background_ratio /= 100;
        vm_dirty_ratio *= correction;
        vm_dirty_ratio /= 100;

        if (dirty_background_ratio <= 0)
            dirty_background_ratio = 1;
        if (vm_dirty_ratio <= 0)
            vm_dirty_ratio = 1;
    }
    mod_timer(&wb_timer, jiffies + dirty_writeback_interval);
    set_ratelimit();
    register_cpu_notifier(&ratelimit_nb);
}
```

page_writeback_init 函数首先获得内存区内所有可分配内存的总数。这里的可分配内存包括 ZONE_HIGH、ZONE_NORMAL、ZONE_DMA 三个管理区。

然后检查当前系统的内存使用情况。buffer_pages 是 ZONE_DMA 和 ZONE_NORMAL 管理区的可分配内存。如果 buffer_pages 少于所有可分配内存 total_pages 的 1/4，此时脏页比例和背景脏页比例两个参数要调整，调整算法是比例参数乘以内存之比。通过放大比例参数达到尽快回写数据回收内存的目的。



注意

此处涉及内核内存管理和分配的知识。简单说，32 位操作系统的内核将内存划分为三个管理区，分别是 ZONE_DMA、ZONE_NORMAL、ZONE_HIGH。

set_ratelimit 函数设置内存页面限制。这个参数的上限是 4 兆内存，按 4K 的页面计算，限制在 16 个页面和 1024 个页面之间，具体根据 CPU 个数和可分配内存数决定。如果页面尺寸不是 4K，就要按 4 兆内存除以实际页面尺寸计算真正的页面最大值。

page_writeback_init 函数启动定时器 wb_timer 作为回写定时器。这个定时器如代码清单 12-3 所示。

代码清单 12-3 wb_timer_fn 函数

```
static DEFINE_TIMER(wb_timer, wb_timer_fn, 0, 0);
static void wb_timer_fn(unsigned long unused)
{
    if (pdflush_operation(wb_kupdate, 0) < 0)
        mod_timer(&wb_timer, jiffies + HZ); /* delay 1 second */
}
int pdflush_operation(void (*fn)(unsigned long), unsigned long arg0)
{
    unsigned long flags;
    int ret = 0;
```

```
    BUG_ON(fn == NULL);     /* Hard to diagnose if it's deferred */

    spin_lock_irqsave(&pdflush_lock, flags);
    if (list_empty(&pdflush_list)) {
        spin_unlock_irqrestore(&pdflush_lock, flags);
        ret = -1;
    } else {
        struct pdflush_work *pdf;
        pdf = list_entry(pdflush_list.next, struct pdflush_work, list);
        list_del_init(&pdf->list);
        if (list_empty(&pdflush_list))
            last_empty_jifs = jiffies;
        pdf->fn = fn;
        pdf->arg0 = arg0;
        wake_up_process(pdf->who);
        spin_unlock_irqrestore(&pdflush_lock, flags);
    }
    return ret;
}
```

定时器 wb_timer 的执行函数是 wb_timer_fn。后者调用 pdflush_operation 函数，从 pdflush_list 全局链表获得一个 pdflush_work 结构，设置 pdflush_work 结构的工作函数 wb_kupdate，然后把 pdflush_work 结构从链表中删除，唤醒 pdflush 内核线程。

pdflush 线程是内核启动的一个内核线程，它的执行函数体是 pdflush。这个函数的作用是执行 pdflush_work 结构设置的工作函数（就是注册的 wb_kupdate 函数）。

12.3.2 执行回写操作

wb_kupdate 是内核回写的控制函数，我们从这个函数开始分析回写机制，如代码清单 12-4 所示。

代码清单 12-4 wb_kupdate 函数

```
static void wb_kupdate(unsigned long arg)
{
    ……/* 省略部分代码 */
    oldest_jif = jiffies - dirty_expire_interval;
    start_jif = jiffies;
    next_jif = start_jif + dirty_writeback_interval;
    nr_to_write = global_page_state(NR_FILE_DIRTY) +
                  global_page_state(NR_UNSTABLE_NFS) +
                  (inodes_stat.nr_inodes - inodes_stat.nr_unused);
    while (nr_to_write > 0) {
        wbc.encountered_congestion = 0;
        /* 一次写页面的最大值为 1024*/
        wbc.nr_to_write = MAX_WRITEBACK_PAGES;
        writeback_inodes(&wbc);
        if (wbc.nr_to_write > 0) {
```

```
            if (wbc.encountered_congestion)
                blk_congestion_wait(WRITE, HZ/10);
            else
                break;      /* All the old data is written */
        }
        nr_to_write -= MAX_WRITEBACK_PAGES - wbc.nr_to_write;
    }
    /* 如果当前时间加 1 秒已超过下一轮的定时器时间，设置定时器为当前时间加 1 秒 */
    if (time_before(next_jif, jiffies + HZ))
        next_jif = jiffies + HZ;
    /* 更新定时器 */
    if (dirty_writeback_interval)
        mod_timer(&wb_timer, next_jif);
}
```

wb_kupdate 函数首先计算三个时间。

- start_jif：当前时间。
- oldest_dif：脏页面允许存在的最长时间，早于这个时间的页面必须回写。这个值默认是当前时间减去 30 秒。
- next_jif：下一次执行回写的时间。默认是 5 秒，就是每 5 秒触发一次 wb_kupdate。

其次检查系统内有多少可写的页面。如果大于 0，则调用 writeback_inodes 执行回写。

执行一次回写操作后，如果还有页面未写并且设置了阻塞标志，则启动阻塞处理。阻塞处理是等 1/10 秒，然后继续执行下一次回写。如果未设置阻塞标志，则直接进行下一轮的回写。

12.3.3 检查需要回写的页面

writeback_inodes 函数用于扫描系统的超级块，检查需要回写的页面，如代码清单 12-5 所示。

代码清单 12-5 writeback_inodes 函数

```
writeback_inodes(struct writeback_control *wbc)
{
  struct super_block *sb;

  might_sleep();
  spin_lock(&sb_lock);
restart:
  sb = sb_entry(super_blocks.prev);
  for (; sb != sb_entry(&super_blocks); sb =
              sb_entry(sb->s_list.prev)) {
      if (!list_empty(&sb->s_dirty) || !list_empty(&sb->s_io)) {
          /* we're making our own get_super here */
          sb->s_count++;
          spin_unlock(&sb_lock);
          if (down_read_trylock(&sb->s_umount)) {
```

```
        if (sb->s_root) {
            spin_lock(&inode_lock);
            sync_sb_inodes(sb, wbc);
            spin_unlock(&inode_lock);
        }
……/* 省略部分代码 */
```

writeback_inodes 函数首先遍历系统的超级块对象，如果超级块的脏页面链表非空，则调用 sync_sb_inodes 回写超级块内的 inode。

12.3.4 回写超级块内的 inode

因为每个文件系统有一个超级块，而文件系统内所有的 inode 结构都链接到超级块对象的链表头，sync_sb_inodes 函数要检查超级块对象内所有需要回写的 inode，如代码清单 12-6 所示。

代码清单 12-6 sync_sb_inodes 函数

```
static void
sync_sb_inodes(struct super_block *sb, struct writeback_control *wbc)
{
  const unsigned long start = jiffies;    /* livelock avoidance */
  /* 将 dirty 链表合并到 s_io 链表 */
  if (!wbc->for_kupdate || list_empty(&sb->s_io))
      list_splice_init(&sb->s_dirty, &sb->s_io);
  while (!list_empty(&sb->s_io)) {
      struct inode *inode = list_entry(sb->s_io.prev, struct inode, i_list);
      struct address_space *mapping = inode->i_mapping;
      struct backing_dev_info *bdi = mapping->backing_dev_info;
      long pages_skipped;

      /* 无回写能力的处理 */
      if (!bdi_cap_writeback_dirty(bdi)) {
          list_move(&inode->i_list, &sb->s_dirty);
          if (sb == blockdev_superblock) {
              continue;
          }
          break;
      }

      /*bdi 指示写拥塞。设置拥塞标志 */
      if (wbc->nonblocking && bdi_write_congested(bdi)) {
          wbc->encountered_congestion = 1;
          if (sb != blockdev_superblock)
              break;    /* Skip a congested fs */
          list_move(&inode->i_list, &sb->s_dirty);
          continue;     /* Skip a congested blockdev */
      }
```

```
        if (wbc->bdi && bdi != wbc->bdi) {
            if (sb != blockdev_superblock)
                break;    /* fs has the wrong queue */
            list_move(&inode->i_list, &sb->s_dirty);
            continue;     /* blockdev has wrong queue */
        }
```

sync_sb_inodes 函数第一部分遍历超级块链表的脏 inode 结构，检查是否可回写。

根据第 9 章块设备文件系统的分析，所有块设备的超级块是块设备文件系统提供的全局变量 blockdev_superblock。

- ❑如果 inode 不具备回写能力，且超级块等于 blockdev_superblock，说明这个 inode 是一个块设备文件的 inode 结构，这种情况继续遍历块设备文件系统的其他 inode，检查是否可写。
- ❑如果超级块不等于 blockdev_superblock，说明这是一个文件系统，这种情况跳过整个超级块，检查下一个超级块里面的 inode。
- ❑如果写拥塞，和上面的处理类似。
- ❑如果文件系统拥塞，跳过整个超级块，检查下一个超级块。
- ❑如果块设备拥塞，只跳过该块设备，检查超级块链表的下一个块设备。

2. 检查时间参数

sync_sb_inodes 函数第二部分首先检查时间参数。

- ❑如果 inode 结构置脏的时间在 sync_sb_inodes 函数开始执行的时间之后，跳出循环。
- ❑如果 inode 结构置脏的时间早于设定的回写时间（回写时间默认是当前时间之前 30 秒），同样跳出循环。
- ❑如果有另外一个 pdflush 线程也在回写队列，同样跳出循环。

```
/*inode 的 dirty 时间在 sync_sb_inodes 调用之后 */
if (time_after(inode->dirtied_when, start))
  break;

/* Was this inode dirtied too recently? */
if (wbc->older_than_this && time_after(inode->dirtied_when,
              *wbc->older_than_this))
  break;

/* 是否已经有另一个 pdflush 在回写这个 inode*/
if (current_is_pdflush() && !writeback_acquire(bdi))
  break;

BUG_ON(inode->i_state & I_FREEING);
__iget(inode);
pages_skipped = wbc->pages_skipped;
__writeback_single_inode(inode, wbc);
……/* 省略部分代码 */
```

3. 检查 inode 状态

如果检查都通过，调用 __writeback_single_inode 检查 inode 的状态，如代码清单 12-7 所示。

代码清单 12-7 __writeback_single_inode 函数

```
static int
__writeback_single_inode(struct inode *inode, struct writeback_control *wbc)
{
  wait_queue_head_t *wqh;

  if (!atomic_read(&inode->i_count))
      WARN_ON(!(inode->i_state & (I_WILL_FREE|I_FREEING)));
  else
      WARN_ON(inode->i_state & I_WILL_FREE);

  if ((wbc->sync_mode != WB_SYNC_ALL) && (inode->i_state & I_LOCK)) {
      list_move(&inode->i_list, &inode->i_sb->s_dirty);
      return 0;
  }

  if (inode->i_state & I_LOCK) {
      DEFINE_WAIT_BIT(wq, &inode->i_state, __I_LOCK);
      wqh = bit_waitqueue(&inode->i_state, __I_LOCK);
      do {
          spin_unlock(&inode_lock);
          __wait_on_bit(wqh, &wq, inode_wait, TASK_UNINTERRUPTIBLE);
          spin_lock(&inode_lock);
      } while (inode->i_state & I_LOCK);
  }
  return __sync_single_inode(inode, wbc);
}
```

__writeback_single_inode 函数执行两个条件判断。

- 如果 inode 状态已经是 LOCK，且回写模式不是 WB_SYNC_ALL，把 inode 链接到超级块的 s_dirty 链表，然后退出。
- 如果回写模式是 WB_SYNC_ALL，一直等待，直到 inode 解除 LOCK 状态，然后调用 __sync_single_inode 函数进行文件的回写。

4. 文件回写操作

__sync_single_inode 函数如代码清单 12-8 所示。

代码清单 12-8 __sync_single_inode 函数

```
static int
__sync_single_inode(struct inode *inode, struct writeback_control *wbc)
{
  unsigned dirty;
```

```
struct address_space *mapping = inode->i_mapping;
struct super_block *sb = inode->i_sb;
int wait = wbc->sync_mode == WB_SYNC_ALL;
int ret;

BUG_ON(inode->i_state & I_LOCK);

/* Set I_LOCK, reset I_DIRTY */
dirty = inode->i_state & I_DIRTY;
inode->i_state |= I_LOCK;
inode->i_state &= ~I_DIRTY;
spin_unlock(&inode_lock);

/* 写页面 */
ret = do_writepages(mapping, wbc);

/* Don't write the inode if only I_DIRTY_PAGES was set */
if (dirty & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) {
    int err = write_inode(inode, wait);
    if (ret == 0)
        ret = err;
}
/* 如果wait设置，则一直等待写硬盘完成 */
if (wait) {
    int err = filemap_fdatawait(mapping);
    if (ret == 0)
        ret = err;
}
```

__sync_single_inode 函数第一部分执行脏数据和 inode 本身的回写。为了表示 inode 数据的各种置脏条件，内核定义了三个参数。

❑I_DIRTY_SYNC：表示只是 inode 访问时间发生了变化。

❑I_DIRTY_DATASYNC：表示 inode 其他属性数据发生了变化，比如文件的长度。

❑I_DIRTY_PAGES：文件的内容数据发生了变化。

__sync_single_inode 函数首先调用 do_writepages 将所有置脏的数据页面进行回写（根据控制参数 wbc 设置的条件）。

如果 inode 设置了 I_DIRTY_SYNC 或者 I_DIRTY_DATASYNC，说明不只是文件的内容，文件本身的属性也发生了改变，因此调用 write_inode 函数对 inode 本身进行回写。当 write_inode 返回时，已经完成了 inode 的回写或者是发生了错误。

如果控制参数 wbc 设置了 WB_SYNC_ALL，说明是紧要数据，需要等所有的数据回写完成。因此调用 filemap_fdatawait 等待数据回写完成。

__sync_single_inode 函数第二部分首先检查文件是否有数据未进行回写。

```
spin_lock(&inode_lock);
inode->i_state &= ~I_LOCK;
if (!(inode->i_state & I_FREEING)) {
```

```
        if (!(inode->i_state & I_DIRTY) &&
            mapping_tagged(mapping, PAGECACHE_TAG_DIRTY)) {
            if (wbc->for_kupdate) {
                /* 让 inode 得到更多的回写机会 */
                inode->i_state |= I_DIRTY_PAGES;
                list_move_tail(&inode->i_list, &sb->s_dirty);
            } else {
                /* 修改 inode 的置脏时间，让其他 inode 有更多的回写机会 */
                inode->i_state |= I_DIRTY_PAGES;
                inode->dirtied_when = jiffies;
                list_move(&inode->i_list, &sb->s_dirty);
            }
        } else if (inode->i_state & I_DIRTY) {
            list_move(&inode->i_list, &sb->s_dirty);
        } else if (atomic_read(&inode->i_count)) {
            list_move(&inode->i_list, &inode_in_use);
        } else {
            list_move(&inode->i_list, &inode_unused);
        }
    }
    /* 唤醒阻塞的进程 */
    wake_up_inode(inode);
    return ret;
}
```

这一步通过调用 mapping_tagged 检查文件的 page cache 状态。

- 如果仍有置脏的数据页面，要将 inode 状态设置 I_DIRTY_PAGES 标志位。
- 如果没有其他的置脏的数据页面，但是 inode 本身被置脏，说明有别的任务修改了 inode 的属性，此时要将 inode 加入超级块的脏 inode 链表。
- 如果以上条件都不成立且 inode 引用计数不为 0，说明文件没有置脏的数据页面，文件属性也没发生改变，将 inode 加入 inode_in_use 链表，如果引用计数为 0，则将 inode 加入 inode_unused 链表。

12.4 平衡写

平衡写在文件系统写过程的上下文执行。第 10 章文件系统读写分析已经观察到写过程调用了 balance_dirty_pages_ratelimited 函数。这个函数作用是检查脏页面是否到达限值，是否进行平衡写操作。具体如代码清单 12-9 所示。

代码清单 12-9 balance_dirty_pages_ratelimited 函数

```
balance_dirty_pages_ratelimited(struct address_space *mapping)
{
  balance_dirty_pages_ratelimited_nr(mapping, 1);
}
void balance_dirty_pages_ratelimited_nr(struct address_space *mapping,
                    unsigned long nr_pages_dirtied)
```

```
{
  static DEFINE_PER_CPU(unsigned long, ratelimits) = 0;
  unsigned long ratelimit;
  unsigned long *p;

  ratelimit = ratelimit_pages;
  if (dirty_exceeded)
      ratelimit = 8;
  /*
   * Check the rate limiting. Also, we do not want to throttle real-time
   * tasks in balance_dirty_pages(). Period.
   */
  preempt_disable();
  p =  &__get_cpu_var(ratelimits);
  *p += nr_pages_dirtied;
  if (unlikely(*p >= ratelimit)) {
      *p = 0;
      preempt_enable();
      balance_dirty_pages(mapping);
      return;
  }
  preempt_enable();
}
```

12.4.1 检查直接回写的条件

balance_dirty_pages_ratelimited 函数调用了 balance_dirty_pages_ratelimited_nr 函数。后者首先定义了一个 CPU 变量 ratelimits，初始化为 0。每次写页面的时候，这个变量累加脏页面的总量，当脏页面超过前面定义的内存页面限制值，则调用 balance_dirty_pages 开始执行回写操作。balance_dirty_pages 函数如代码清单 12-10 所示。

代码清单 12-10 balance_dirty_pages 函数

```
static void balance_dirty_pages(struct address_space *mapping)
{
  ……/* 省略部分代码 */
  for (;;) {
      struct writeback_control wbc = {
          .bdi            = bdi,
          .sync_mode      = WB_SYNC_NONE,
          .older_than_this = NULL,
          .nr_to_write    = write_chunk,
          .range_cyclic   = 1,
      };

      get_dirty_limits(&background_thresh, &dirty_thresh, mapping);
      nr_reclaimable = global_page_state(NR_FILE_DIRTY) +
                       global_page_state(NR_UNSTABLE_NFS);
```

```
        if (nr_reclaimable + global_page_state(NR_WRITEBACK) <= dirty_thresh)
                break;
        if (!dirty_exceeded)
            dirty_exceeded = 1;
```

balance_dirty_pages 函数第一部分首先调用 get_dirty_limits 获得设置的脏页面回写数目。前面通过 page_writeback_init 函数已经设置了系统的脏页面比例。变量 background_thresh 和 dirty_thresh 分别代表背景写的脏页面数目和直接回写的脏页面数目。

12.4.2 回写系统脏页面的条件

然后计算系统内需要回写的脏页面数目总和。如果这些页面总数超过 dirty_thresh 的限制，则进入直接回写，否则退出循环，检查是否超过了 background_thresh 限制。

```
if (nr_reclaimable) {
  writeback_inodes(&wbc);
  get_dirty_limits(&background_thresh,
                    &dirty_thresh, mapping);
  nr_reclaimable = global_page_state(NR_FILE_DIRTY) +
                   global_page_state(NR_UNSTABLE_NFS);
  if (nr_reclaimable +
          global_page_state(NR_WRITEBACK)       <= dirty_thresh)
      break;
  pages_written += write_chunk - wbc.nr_to_write;
  if (pages_written >= write_chunk)
      break;    /* We've done our duty */
  }
  blk_congestion_wait(WRITE, HZ/10);
  }

  if (nr_reclaimable + global_page_state(NR_WRITEBACK)
      <= dirty_thresh && dirty_exceeded)
      dirty_exceeded = 0;
  /* 如果 pdflush 线程已经在写当前队列，退出 */
  if (writeback_in_progress(bdi))
      return;    /* pdflush is already working this queue */
  /* 如果是笔记本模式，要等达到直接写的标准才开始背景写 */
  if ((laptop_mode && pages_written) ||
      (!laptop_mode && (nr_reclaimable > background_thresh)))
      pdflush_operation(background_writeout, 0);
}
```

balance_dirty_pages 函数第二部分调用 writeback_inodes 回写系统的脏页面。回写完毕，要执行两个检查：一是检查脏页和回写页是否大于 dirty_thresh 脏页限制，二是检查已经回写的页面是否小于设置的回写页面数，如果两个条件都满足，则阻塞进程 1/10 秒，然后再次回写。

12.4.3 检查计算机模式

最后，balance_dirty_pages 函数要检查计算机的模式。

- 笔记本模式：这种模式回写的标准比较高，要等待达到标准较高的直接写条件，才进行背景写。
- 普通模式：检查脏页面是否超过背景写的限制数 background_thresh，如果超过，触发 pdflush 线程进行回写。

writeback_inodes 前文已经分析过，不再赘述。

12.5 本章小结

Linux 内核的回写机制提供了一种缓写机制，真实的写并不是直接写入硬盘，而是在 page cache 中缓存，等待合适的时机才真正执行写入。这种机制存在的前提是适配硬盘的物理特性，在内核软件的设计中，乃至用户软件的设计中，考虑硬件特性并适配之，是系统设计的重要方面。

第 13 章

一个真实文件系统 ext2

通过前面章节的分析和学习，本章分析一个真正的文件系统。本章选的例子是 Linux 的 ext2。ext2 是 Linux 系统使用最广泛的文件系统，它和 ext3 构成 Linux 文件系统的基石。而 ext3 和 ext2 的结构基本相同，只是 ext3 多了日志功能。本章不再分析代码，主要介绍 ext2 的磁盘布局和超级块，以及 inode 结构，代码分析工作留给读者。原因是 Linux 内核庞大复杂无比，任何书都不可能完备的分析代码。通过本书内核基础层和应用层的分析，读者可以从架构层次掌握系统的框架和脉络，在这个基础上，可以比较流畅的分析阅读内核代码。

13.1　ext2 的硬盘布局

根据 Linux 文件系统的知识，文件和 dentry 是内存中的结构，并不真正存在于硬盘中。真正存在于硬盘的是超级块结构和 inode 结构。首先看 ext2 的硬盘布局，如图 13-1 所示。

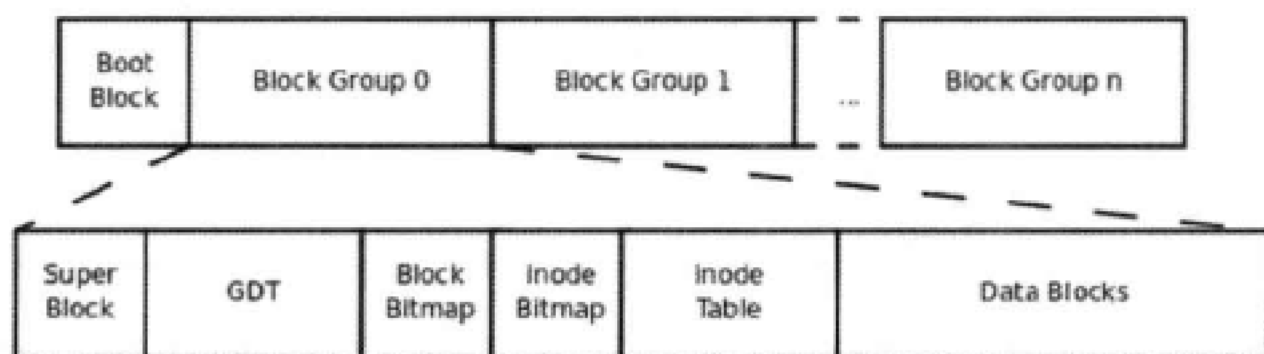


图 13-1　ext2 硬盘布局

硬盘的第一个扇区是引导区，占据 1K 字节，引导区是文件系统不能使用的，用来存储分区信息。ext2 是通过块组的方式来组织硬盘。每个硬盘分区都由若干个大小相同的块组组成。每个块组包括下列信息。

- 超级块（super block）：每个块组的起始位置都有一个超级块，这些超级块的内容都是相同的。超级块包括文件系统的信息，比如每个块组的块数目、每个块组的 inode 数目等。
- 块组描述符表（GDT）：块组描述符表由很多的块组描述符组成。ext2 的每个块组描

述符为 32 字节。整个文件系统分区有多少个块组，就有多少个块组描述符。块组描述符描述了一个块组的信息，比如一个块组中 inode 位图的起始位置、inode 表的起始位置等。

- 块位图（block bitmap）：每个比特代表块组中哪些块可用，哪些块已经被占有。块位图本身要占有一个块。如果块大小设置为 1K 字节，则一个块组的大小为 1K × 1K × 8bit = 8M 字节。
- inode 位图（inode bitmap）：inode 位图也占用一个块。它的每一位代表一个 inode 是否空闲。
- inode 表（inode table）：每个文件都有一个 inode，inode 保存了文件的描述信息、文件的类型、文件的大小、文件的创建访问时间等。一个 inode 占 128 字节。如果文件系统块大小为 1K 字节，一个块可容纳 8 个 inode。inode 表可以占用多个块。
- 数据块（data block）：保存文件的内容。常规文件的数据保存在数据块中，如果是目录文件，那么该目录下所有的文件名和下级目录名都保存在数据块中。

13.2 ext2 文件系统目录树

文件系统的层次结构可以用目录树来表示。对 ext2 文件系统而言，根目录是一个固定的 inode。通过读根目录的内容，就可以获得根目录下的文件的 inode 信息、文件类型和文件名字。如果该文件是目录文件，重复这个过程，就可以获得二级目录的信息。

ext2 目录文件的结构如代码清单 13-1 所示。

代码清单 13-1 ext2_dir_entry_2

```
struct ext2_dir_entry_2 {
  __le32   inode;                /* Inode number */
  __le16   rec_len;              /* Directory entry length */
  __u8     name_len;             /* Name length */
  __u8     file_type;
  char     name[EXT2_NAME_LEN];  /* File name */
};
```

根目录是一个固定的 inode，它的 inode 号是 2。当文件系统初始化的时候，读到超级块的内容，就获得了文件块的大小、每个块组的块数目、每个块组的 inode 数目。从超级块之后的块组描述符表，可以获得所有的块组信息。

通过超级块提供的信息和块组描述符表信息，就可以获得根目录在 inode 表的位置，从而读到根目录的内容。从根目录的内容里面，进而获得根目录下文件的 inode 号和文件类型。

如果这是个目录文件，那么重复上面的过程，可以获得二级目录的目录结构。不断重复这个过程，最终获得整个文件系统的目录树。

13.3 ext2 文件内容管理

ext2 的 inode 信息可以存放 15 个块的地址，用户数据就存放在这些块中。但是 15 个块不一定能存放全面的用户数据，那么可以采用分层的方式存储。这 15 个块中，前 12 个块是直接数据块，直接存放用户数据。而第 13 个块，是一级索引块，这个块存放的又是块的地址，这些块地址指向的块才真正存放用户数据。而第 14 个块，是二级索引块，比一级索引块又多了一层，而第 15 个块，是三级索引块，比二级索引块又多了一层。通过这种方式，大大扩展了 ext2 文件系统的文件存放内容。

13.4 ext2 文件系统读写

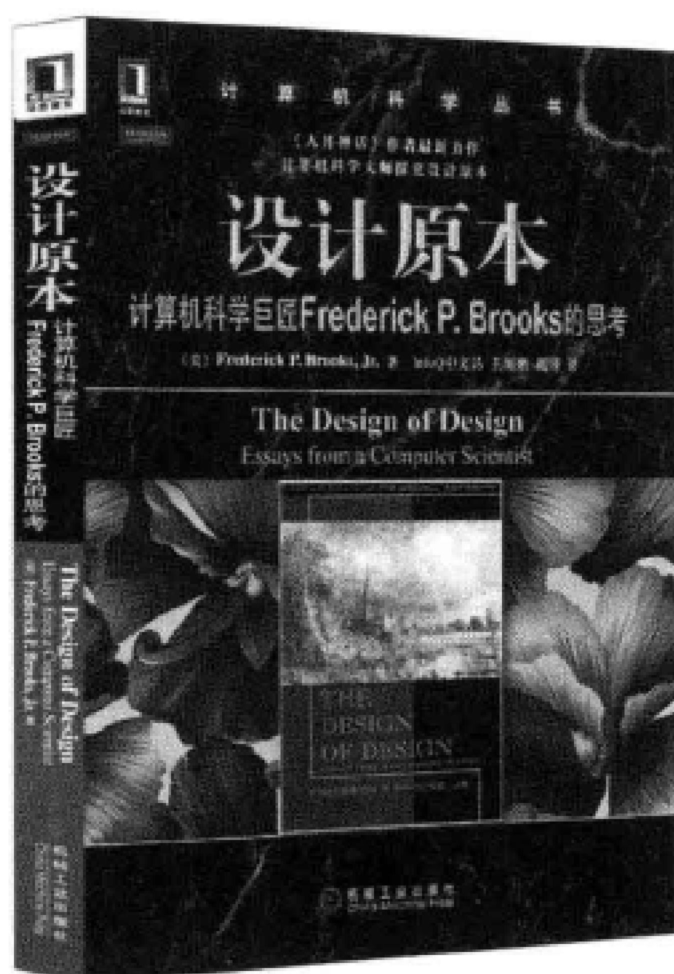
对一个文件系统来说，最重要基础的部分无非是打开和创建文件以及文件的读写。

通过文件系统的目录树，可以获得文件所在目录的上级目录。读上级目录的内容，就可以获得文件的 inode 号和文件名字。根据文件的 inode 号，可以获得文件的 inode 信息。也就获得了文件的类型、创建修改时间、文件大小等信息，完成打开文件的过程。

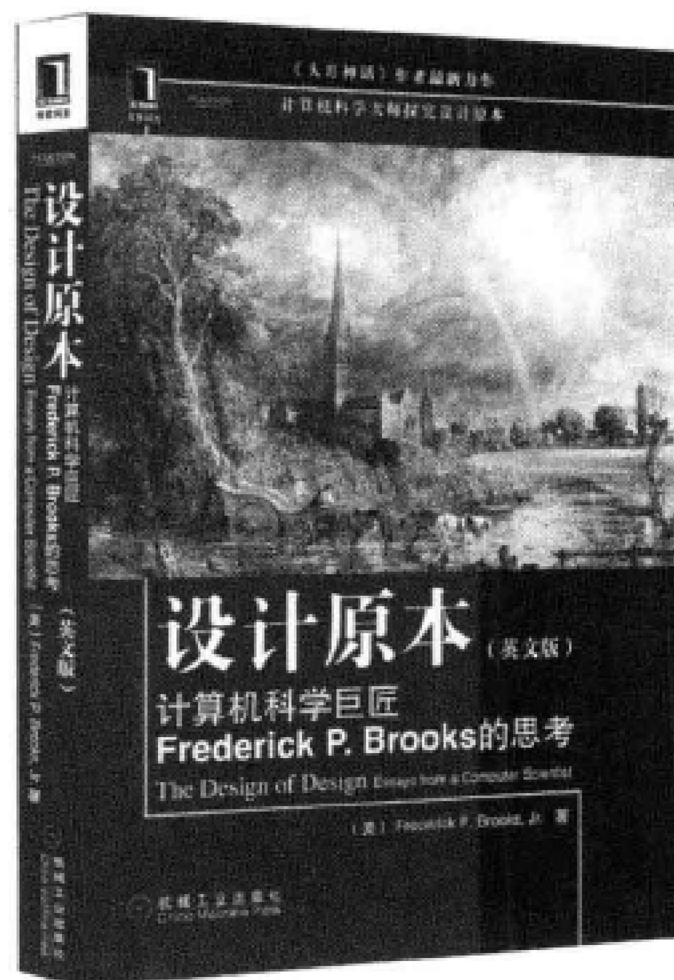
文件的读写，首先要获得文件数据内容的物理扇区位置。在文件的 inode 信息里，保存了 15 个文件块的地址。这 15 个文件块通过分层方式，存储了文件的数据内容。知道了文件数据的块地址，就可以获得文件数据的物理扇区地址。根据第 10 章的分析，可以真正对文件执行读写操作。

13.5 本章小结

本章可以作为对内核学习的一个实战。读者应该结合 ext2 文件系统的代码，分析文件系统的读写和打开的过程，深度理解一个广泛使用的文件系统。内核本身在不断更新，想利用内核知识解决实际问题，必须具备快速阅读代码的能力，从代码中学习，从代码中实践。



计算机科学丛书
设计原本
计算机科学巨匠Frederick P. Brooks的思考
The Design of Design
Essays from a Computer Scientist



设计原本（英文版）
计算机科学巨匠
Frederick P. Brooks的思考
The Design of Design